VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÍ PROGRAMOVACÍHO JAZYKA C PLUS A JEHO PŘEKLADAČE

AN EXTENSION OF THE C PLUS PROGRAMMING LANGUAGE AND ITS COMPILER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETR OPATŘIL**
AUTHOR

VEDOUCÍ PRÁCE **Prof. RNDr. ALEXANDER MEDUNA, CSc.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20192/2016/xopatr00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Opatřil Petr, Bc.**
Obor: Inteligentní systémy
Téma: **Rozšíření programovacího jazyka C Plus a jeho překladače**
An Extension of the C Plus Programming Language and Its Compiler
Kategorie: Překladače

Pokyny:

1. Podrobně se seznamte s programovacím jazykem C Plus, který byl zaveden v práci 2 (viz Literatura níže). Dále se seznamte s vybranými metodami syntaktické analýzy, návrhem a konstrukcí překladačů dle instrukcí vedoucího.
2. Dle instrukcí vedoucího nastudujte a proveďte srovnání předností populárních programovacích jazyků, zejména oblast generického programování a metaprogramování.
3. Získané poznatky z bodu 2 aplikujte návrhem vhodných rozšíření pro jazyk C Plus.
4. Implementujte překladač navrhnutého jazyka z bodu 3.
5. Ve vytvořeném programovacím jazyce implementujte nejméně 10 programů demonstrujících možnosti navrhnutého jazyka.
6. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

1. Meduna, A.: Elements of Compiler Design, Taylor & Francis, New York, 2008
2. Opatřil, P.: Rozšíření programovacího jazyka C a jeho překladače, bakalářská práce, FIT VUT, Brno, 2014
3. Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práce se zabývá pokračujícím vývojem nového programovacího jazyka C Plus navrženého v předchozí balakářské práci s cílem rozšířit jazyk C o vybrané vysokoúrovňové techniky bez přidané režie. V rámci práce byla srovnána řada jazyků, C Plus i s jeho gramatikou byly obohaceny o řadu nové funkcionality a byly diskutovány přínosy a srovnání s realizací v konkurenčních jazycích. Navržená rozšíření byla implementována v překladači.

Abstract

This thesis describes continuing development of new programming language C Plus conceived in earlier Bachelor's Thesis oriented on enhancing C language with high level constructs with no additional cost. During development, several important languages were compared and C Plus along with its grammar were expanded, advantages of additions were discussed and compared with solutions in other languages. Described enhancements were implemented in compiler.

Klíčová slova

Jazyk C Plus, rozšíření C, překladače, C+

Keywords

C Plus language, C extension, compilers, C+

Citace

OPATŘIL, Petr. *Rozšíření programovacího jazyka C Plus a jeho překladače*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Meduna Alexander.

Rozšíření programovacího jazyka C Plus a jeho překladače

Prohlášení

Prohlašuji, že jsem tento projekt vypracoval samostatně pod vedením pana prof. RNDr. A. Meduny, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Opatřil
23. května 2017

Obsah

Kapitola 1

Úvod

Tématem této práce je pokračující vývoj nového programovacího jazyka s názvem C Plus (dále zkracované na *C+*), tvořící vysokoúrovňovou nadmnožinu programovacího jazyka C. O důvodech tvorby a výhodách tohoto jazyka pojednává následující kapitola, stručně ho však lze charakterizovat jako sadu rozšíření zvyšující uživatelský komfort a celkovou čitelnost programů, snažící se ale vyhnout nárůstu provozní režie, který je s podobným vývojem typický.

Tento programovací jazyk byl vytvořen jako produkt bakalářské práce z roku 2015. Kromě návrhu jazyka samotného byl vyvinut i jeho překladač. Ten překládá C+ do čistého C, které je vzhledem ke své rozšířenosti široce použitelné a tranzitivně tak jazyku zajišťuje vysokou výkonnost. [22]

V rámci bakalářské práce byl tento překladač uspěšně uveden do provozu. Kromě významných rozšíření přicházejících s C+ má kompletní podporu pro syntaktickou a sémantickou analýzu C standardu C99 [17], což se paradoxně rozsahem a náročností ukázalo jako nejvýznamnější část projektu. Z časových důvodů však nebyla vyřešena podpora *preprocessoru* – svázaného, ale jinak zcela ortogonálního jazyka pracujícího na principu textových substitucí. Aby tedy nebyl překladač použitelný pouze akademicky, bylo nutné prvně vyřešit tento problém a plně tak automatizovat cestu od zdrojového kódu v C+ ke spustitelnému souboru.

Kromě toho byl jazyk obohacen o řadu rozšíření. Jde jednak o funkcionalitu po konceptuální stránce rozpracovanou do jisté míry už v době BP, například reference. Jiná rozšíření jsou zcela nová a byly inspirovány studií charakteristik populárních programovacích jazyků – kromě C++ a Pythonu, které významně ovlivnili jazyk od počátku, byl průzkum rozšířen i o vybrané části Javy, C#, Objective-C a třeba i minimalističností Haskellu.

Ačkoli se jazyk C+ nehlasí k objektově orientovanému paradigmatu, ukázalo se výhodné adoptovat některé konvenční vlastnosti OO jazyků. Jde jak o možnost lépe svázat funkce s konkrétním datovým typem a vytvářet tím metody, tak např. přidání konstruktorů a destruktorů, umožnující datové typy automaticky uvolňující zdroje.

Cílem práce však samozřejmě není pouhé kopírování funkcionality odjinud. Ačkoli je pravda, že na některých prvcích už nebylo co vylepšit, u jiných analýza odhalila potenciál ke zlepšení – jde např. o problémy se syntaktickou nejednoznačností volání konstruktorů v C++ či explicitnější realizace tzv. přesouvací (*move*) sémantiky přímo v gramatice jazyka, vedoucí na vyšší optimalizační potenciál.

Kapitola 2 do detailů představuje vytvářený jazyk, jeho základní ideologii a komponenty. Dále rekapituluje již realizované části jazyka a představuje LL gramtatiku pro něj vytvořenou se zdůvodněním její významnosti.

Kapitola 3 se věnuje popisu a srovnání konkrétních programovacích jazyků, které ovlivňili vývoj, jejich specifik a silných a slabých stránek z pohledu adaptace funkcionality do C+.

Další kapitola 4 rozebírá jednotlivá nová rozšíření – motivace k jejich tvorbě, rozbor konkurenčních realizací, přínos vlastního řešení a komplikace, které nastaly při implementaci.

Kapitola 5 popisuje ostatní vykonanou práci na překladači, která nutně nespadá pod rozšíření popsaná dříve.

Konečně, kapitola 6 shrnuje dosažené výsledky.

Diplomová práce navazuje na předchozí Semestrální projekt, ten již informoval o částech práce v té době dokončených (zejména jde o obsah kapitol „Automatizace procesu překladu“, „Objektově orientovaná rozšíření“ a „Reference jako alternativa ukazatelů“). Všechny úseky, kde je to adekvátní, tak z projektu přebírá. Kromě tohoto je však text diplomové práce originální.

Kapitola 2

Seznámení s C+

2.1 Motivace a ideologie jazyka

Jazyk C+ je strukturovaný, staticky typovaný a imperativní programovací jazyk se zpětnou kompatibilitou s C. Jazyk C je jednoznačně jeden z nejpopulárnějších nízkoúrovňových jazyků a díky své popularitě je nejen použitelný na širokém spektru platforem, ale disponuje i velmi efektivními a rychlými překladači. Programy v C jsou tak v testech obvykle rychlejší než libovolný jiný kód až na ručně psané programy v jazyce symbolických instrukcí a i ty dokáže dobrou znalostí konkrétních instrukčních sad často překonat. Bohužel je ale C oproti např. Pythonu výrazně méně komfortní – tedy pro naprogramování stejné funkcionality je nutné napsat podstatně více kódu.

Klíčová myšlenka za vznikem C+ je pozorování, že existuje celá řada zlepšení, která zvyšují uživatelský komfort i celkovou čitelnost, aniž by však vedly k méně efektivnímu programu. V mnoha situacích je možné dosáhnout kompaktnosti zápisu na úrovni jmenovaného Pythonu, s výpočetní rychlostí o několik řádů vyšší a s překladem, který je srovnatelný s dobou spouštění interpretu.

To jazyku otevírá zcela nové oblasti použití, ve kterých bylo běžné psát výkonostně klíčové části kódu ve zkompilované C/C++ knihovně, a vysokoúrovňový koordinační kód v dynamickém skriptovacím jazyku, čistě pro uživatelský komfort / rychlost programování.

Důležitým cílem je i interoperabilita s C. Průzkum jazyků jasně prokázal, že pozice C/C++ coby průmyslového standardu je v současnosti neotřesitelná a důvod, proč se mnoho experimentálních jazyků zatím neujalo, bylo přílišné vzdálení se od syntaxe rodiny C, pro programátory dobře známé. Kromě časové investice do zvládnutí nuancí jazyka je tu také druhý faktor – pro klasické jazyky existuje řada vývojových a podpůrných nástrojů, mnohdy celé vývojové frameworky. Mimo jiné kvůli tomuto je C+ vyvýjeno jako nadmnožina C, což teoreticky umožňuje nejen být bez dalších úprav, či jen s malými změnami zpracovatelný některými ze zmíněných nástrojů, ale také těží z toho, že C hlavičkové soubory jsou obvyklým prostředkem popisujícím binární rozhraní mezi jazyky.

Je také nutné zdůraznit odlišné zacílení oproti C++, které také vzniklo rozšiřováním C, a je typickou volbou pro psaní kódu citlivého na výkon. Kromě toho, mnoho rozšíření C+ z této práce poskytuje funkcionalitu, která už je v C++ k dispozici, zdůvodnění praktičnosti je tak na místě. Zjednodušeně řečeno, C+ pokrývá právě ty případy užití C++, kde se místo něj stále používá C, tedy kód malého rozsahu se silným důrazem na efektivnost.

C++ jako jazyk umožňuje kombinovat více programovacích přístupů, běžně je však používán jako objektově orientovaný jazyk. To má řadu výhod jako skvělou škálovatelnost, modularitu a aplikaci osvědčených praktik softwarového inženýrství, ale pro některé

aplikace je rigidní objektový systém přehnaně komplexní a důsledné zapouzdření často limituje optimalizační potenciál. Tyto argumenty jsou důkladněji rozebírany v sekci 4.2.1, stačí však poznamenat, že na tuto specializaci (tedy systémové programování, vestavěné & real-time systémy apod. bez objektového návrhu) míří i jiné nové jazyky, např. Rust. Od nich se C+ odlišuje tím, že každý z těchto zkoumaných jazyků se snaží *nahradit* C, nikoli ho pouze doplnit a zpříjemnit. Obojí má své výhody, C+ je zpětnou kompatibilitou limitováno v možnostech rozšíření, zato se ale ale zcela bez problémů kombinuje a spolupracuje s kódem v čistém C.

2.2 Konkrétní charakteristiky jazyka

V t0to kapitole jsou představeny některé z rozšíření funkcionality C, které C+ přineslo v rámci BP.

Řídící konstrukce cyklů byly rozšířeny o `else` větve, podobně jako má jazyk Python. Tato větev je navštívena v momentě, kdy podmínka cyklu přestane platit – to může být zdánlivě neužitečné, neboť je tím pádem navštívena skoro vždy, klíčovým rozdílem je chování příklazu `break`, který skáče až za konec cyklu. Toto umožňuje elegantně realizovat případ použití, kdy v cyklu iterujeme nad nějakou strukturou, dokud nenastane vyjímečná podmínka (např. jsme nalezli element v kolekci, případně operace prováděná nad elementy selhala) – `else` větev pak pokrývá situaci, kde iterace skončila bez přerušení. Jak ukazuje následující případ 2.1, jde o pčípad ideálního rozšíření pro tento jazyk – lze implementovat jednoduše a efektivně v C pomocí `goto`, zatímco obvyklé řešení tohoto problému v C by se `goto` z důvodů čitelnosti záměrně vyhnulo[1] a patrně by se zavedla pomocná proměnná, tedy suboptimální řešení.

Z důvodů symetričnosti bylo umožněno použít `else` větev i pro příkaz `switch`, kde se jedná o alternativní, ale ekvivalentní syntaxi k použití návěstí `default`.

```
*initialization*;
while (*condition*) {
  ...
  if (*test*)
    break;
  ...
}
else {
  *else statement*
}
*next statement*
```

Kód v C+

```
*initialization*;
while (*condition*) {
  ...
  if (*test*)
    goto WHILE_END;
  ...
}
{
*else statement*
}
WHILE_END: *next statement*
```

Kód v C

Obrázek 2.1: Ukázka konverze `else` větve cyklu na ekvivalentní kód v čistém C.

Dalším rozšířením je možnost víceúrovňového `break` a `continue`, tedy schopnost aplikovat je i na jiné cykly než ten nejbližší. Toto je velmi významný posun, protože vyskošení z vícenásobných zanořených cyklů je jinak velmi komplikované a jde o jeden z mála případů, kde je `goto` některými považováno za čistější a čitelnější řešení než sekvence `break`ů syn-

[1] Problémy s přehnaným využíváním `goto` shrnul Dijkstra ve svém dnes už klasickém článku, od té doby je normou jeho využití minimalizovat na úkor strukturovaných konstrukcí. [18]

chronizovaná pomocnými proměnnými. Implementace v C+ tento přístup využívá, uživatel je však od něj odstíněn a automatizace zaručuje, že v řešení nevznikne chyba (2.2).

```
maincycle: for (int i=0;i<argc;i++)
{
  bool condition=true;
  while (condition)
  {
    switch (argtype(argv[i]))
    {
      case 1:
        if not (i%2)
          break maincycle;
        else
          continue maincycle;
      case 0:
        function1(&condition);
      break;
    }
    else
    {
      function2();
      condition=false;
    }
    break maincycle;
  }
  break;
}
else
  return 1;
return 0;
```

Kód v C+

```
maincycle: for (int i=0;i<argc;i++)
{
  bool condition=true;
  while (condition)
  {
    switch (argtype(argv[i]))
    {
      case 1:
        if (not(i%2))
          goto FOR_BREAK;
        else
          goto FOR_CONTINUE;
      case 0:
        function1(&condition);
      break;
      default:
        function2();
        condition=false;
    }
    goto FOR_BREAK;
  }
  break;
  FOR_CONTINUE:;
}
return 1;
FOR_BREAK:
return 0;
```

Kód v C

Obrázek 2.2: Ukázka konverze `else` větví a výskok z víceúrovňového cyklu na ekvivalentní kód v čistém C.

U funkcí lze definovat implicitní hodnoty parametrů funkcí. Toto klasické rozšíření známé např. z C++ je velmi výhodné pro ty parametry, co mají obvyklou neměnnou hodnotu, která se mění ve vyjímečných případech. Typickým příkladem je volitelná struktura předávaná ukazatelem – pokud není dostupná, předává se `NULL`. Kromě toho, že takovéto implicitní hodnoty ušetřují programátorovi práci, redukce duplicitního kódu snižuje riziko chyb[2]

Konstrukce `typedef` na rozdíl od sémantiky C/C++, kde je výsledkem pouze jiné pojmenování stejného typu, zde vytváří kompletní, disjunktní datový typ. Tento nový typ je bez dalších úprav implicitně přetypovatelný na původní a tak je zachována zpětná kompatibilita. Mimo to ale pro takový typ lze i předefinovat operátory a je považovaný za unikátní pro účely přetěžování funkcí, prakticky tedy není problém vytvářet vlastní datové typy, které jsou syntakticky stejně pohodlné jako typy nativní – klasický příklad je trojrozměrný vektor čísel typu `double`, pro který dává smysl přetížit aritmetické operátory.

Jak bylo uvedeno dříve, jazyk není objektový, operátory jsou chápány jako libovolné jiné funkce a rozeznávány jsou jen pomocí speciálního jména, např. funkce realizující operátor

[2]Redukce duplicitního kódu je obecně brána jako pozitivní faktor jazyka a někdy je označována jako „Princip DRY“ (Don't Repeat Yourself).

sčítání by se jmenovala `T__add` pro typ T. To, že funkce nejsou součástí definice datového typu, povoluje i další flexibilitu, kdy si další modul může dovolit dodatečně, lokálně dodat datovému typu novou funkcionalitu. Oproti C++ také není takováto činnost limitována na strukturované typy (resp. `class`), ale je obecně bez omezení. Další výhodou toho, že operátory jsou konvenční funkce, je to, že jsou použitelné jako libovolný jiný funkční ukazatel, včetně explicitního volání z jiného jazyka.

Jazyk také povoluje přetěžování funkcí, tedy vytváření množiny funkcí se stejným jménem, lišící se v datových typech parametrů. Toto lze snadno implementovat zcela bez režie, neboť výběr správné volby lze realizovat staticky při překladu. C++ umožnuje i dynamický polymorfismus na základě virtuálních funkcí přetížitelných v podtypu, to vyžaduje režii za běhu programu – C+ však nepodporuje dědičnost a tím i podobnou funkcionalitu. Specialitou C+ je možnost přetěžovat i podle návratového typu, ovšem pro stejné parametry jsou možné pouze varianta przdná (`void`) a neprázdná. To umožňuje zvýšit efektivitu v případech, kde je návratová hodnota často ignorována, jako třeba u knihovní funkce `printf`.

Cílem bylo také analyzovat C syntaxi a odhalit redundance, umožňující kratší zápis, zde bylo dosaženo částečných úspěchů.

Jazyk podporuje volitelně možnost odsazování bílými znaky, tedy vymezení bloků kódu odsazením namísto obvyklých složených závorek. Tato funkcionalita byla inspirována Pythonem a vychází z pozorování, že u každého konzistentně zarovnávaného kódu (což by patrně mělo pokrývat každý kvalitně napsaný program) jsou závorky nadbytečné, neboť úroveň zanoření řádku lze vypozorovat z jeho odstazení. Jde však i o lehký zdroj kontroverze. Někteří poukazují na to, že tento způsob limituje možnosti formátování; pevný, konzistentní styl však může stejně tak být chápan jako přednost a jako čitelnost zvyšující faktor. Další výtky jsou, že programy, které na toto nejsou připravené, mohou snadno zarovnávání narušit, případně že kombinování tabulátorů s mezerami vede k nevalidnímu kódu, který vypadá zcela v pořádku[3]. Konečně, někteří mají striktní námitky ryze estetického/filosofického charakteru proti využití bílých znaků jazykem.

Této kritice se C+ může částečně vyhnout, neboť je tato funkce volitelná – je použita jen na určených místech na explicitní vyžádání programátora, např. tak, že se za podmínkou `if`u umístí dvojtečka (syntakticky tedy záměrně podobné realizaci v Pythonu) a od daného řádku dál pak začíná bílými znaky odsazený blok.

Ukázalo se také, že středníky za příkazy a kulaté závorky pro podmínky řídících konstrukcí jsou až na několik speciálních situací nadbytečné, tedy i po jejich odstranění je kód jednoznačně rozhodnutelný. Většina situací, kdy rozhodnutelný není, nedává praktický smysl, např. vyžaduje následující příkaz tvaru „`-1;`“. Bohužel, dají se najít i příklady kompletně užitečného nejednoznačného kódu, odstranit se tak nedají. Nicméně, díky faktu, že jejich vynechání z pravidla obvykle neudělá kód nepřeložitelný, lze chybějící středník či závorku brát často jen jako malou chybu a klasifikovat ji jako varování.

Pro jazyk byl naprogramován od základů nový překladač v čistém C, který C+ kód převádí na čisté C. Překladač využívá rekurzivní sestup a podporuje jak kompletní C99, tak všechna zmíněná rozšíření.

Pro jazyk byla navržena formální gramatika, oproti běžné LR gramatice jazyků vycházejících z C se ji podařilo zjednodušit na LL(1), což vede na jednoznačnou volbu pravidla jediným tokenem. Bohužel, gramatika se oproti srovnatelné LR verzi výrazně zvětšila, praktičnost takové gramatiky je tak jen omezená, použití pro úspěšnou automatickou konstrukci LL tabulky však umožnilo potvrdit její jednoznačnost. [21]

[3]Programy kombinující více druhů bílých znaků v odsazení jsou považovány za chybu, je však snadno opravitelná s každým kompetentním textovým editorem.

```
int fakt(int x):
  if (x == 1):
    print(x);
    return 1;
  else: print(x);return x*fakt(x-1);
```

Kód v C+

```
int fakt(int x)
{
  if (x == 1)
  {
    print(x);
    return 1;
  }
  else
  {
    print(x);
    return x*fakt(x-1);
  }
}
```

Kód v C

Obrázek 2.3: Ukázka odsazování bílými znaky

```
func(1,2)
*p++

if (x == 3)
  *p--;
```

Obrázek 2.4: Příklady nejasného kódu při uvolnění syntaxe

Kapitola 3

Analýza jazyků

V rámci této diplomové práce bylo nutné prozkoumat a porovnat možnosti řady programovacích jazyků, tato kapitola obsahuje stručnou charakteristiku těch, jejichž vlastnosti byly zvlášť významné, jak pro inspiraci, tak kritiku. Z praktických důvodů zde nejsou rozebírány přílišné detaily, protože relevantní konkrétní vlastnosti jsou obvykle rekapitulovány a rozebírany dopodrobna v kapitolách, které se věnují odpovídajícímu rozšíření. Obecné informace o mechanismech a třídách programovacích jazyků byly čerpány ze zdroje [23].

3.1 Jazyk C

Vzhledem k tomu, že celé téma práce vychází z prozkoumávání potenciálu ke zlepšení a objevování nedostatků rodiny jazyků odvozených z C, od čtenáře se očekává jistá znalost syntaxe a možností tohoto programovacího jazyka.

Jde o klasický strukturovaný, imperativní jazyk používaný od 70. let, začínající s operačními systémy na bázi Unixu. Významným mezníkem byla standardizace ANSI C roku 1989, která je dodnes považována za primární společný základ, podporovaný všemi překladači.

Jazyk se od té doby měnil jen relativně (vůči např. C++) málo, verze z roku 1999 byla zvolena jako základní reference pro překladač C+. C99 přineslo možnost kombinovat deklarace proměnných s příkazy, řádkové komentáře zpětně převzané z C++, nativní komplexní typy i některá rozšíření chybějící v C++, jako pole proměnné délky na zásobníku.

Poslední verze z roku 2011, C11, přináší jen drobnou evoluci jako je podpora multithreadingu a atomik, kde v zásadě jde jen o standardizaci funkcí běžně poskytovaných významnými překladači. Z tohoto důvodu nebyla podpora C11 rozšíření považována při vývoji překladače považována za prioritu, byť je plánována do budoucna.

Významnou roli zde hraje i preprocesor, který konceptuálně předzpracovává zdrojový kód předtím, než je syntakticky analyzován překladačem C, prakticky tak jde o zcela separátní jazyk určený k provádění transformací textu. Oddělenost preprocessoru od zbytku jazyka je natolik významná, že je beze změny použitelný i v jiných jazycích, jako je C++. Dá se říci, že role preprocessoru je dnes už historický artefakt, moderní jazyky nezatížené zpětnou kompatibilitou řeší jeho úlohy (tedy zajištění modularity, vytváření alternativních jmen symbolů, základní typově generický kód, direktivy pro překadač) přímo v jazyce. Využívání preprocessoru k netriviálním úkolům je problematické, např. generické programování pomocí maker má ve srovnání s C++ šablonami obvykle slabší typovou bezpečnost a především zatemňuje sémantiku překladači, který pak nedokáže produkovat čitelná chybová

hlášení. O problémech, které preprocesor přinesl při překladu do C, hlouběji pojednává sekce 4.1.2. [17][3][13]

3.2 C++

C++ je multiparadigmový, primárně však objektově orientovaný jazyk vycházející z C. Až na detaily je zpětně kompatibilní, prioritou při návrhu byl vysoký výpočetní výkon. Oproti C, které obsahuje jen základní množinu jazykových konstrukcí, která jde efektivně mapovat na možnosti strojového jazyka, a poskytuje jen minimální standardní knihovnu, jde o signifikantní rozšíření. Pro rámcové srovnání, standard C11 [13] má 701 stránek, zatímco C++11 [14] má 1334. S každou verzí dochází k významnému rozšiřování standardních knihoven i povolené syntaxe – to sice na jednu stranu zvyšuje sílu jazyka a zvyšuje uživatelský komfort[1], ale vysoká komplexnost jazyka má následující nevýhody:

- Velké množství duplikované funkcionality vznikající tím, že nové rozšíření nahradí případy užití předchozího (např `std::array` vs klasické pole z C), předchozí funkcionalita kvůli zpětné kompatibilitě zůstává. Toto souvisí i s následujícím.
- Dlouhá doba, než se nový programátor seznámí se všemi prostředky jazyka a naučí se je využívat efektivně.
- Složitost jazyka se odráží ve složitosti jeho překladačů. Je poměrně snadné napsat základní překladač pro C od základu, velmi obtížné pro C++. To vede jednak k tomu, že je proces přidávávání nové funkcionality stále kompikovanější, optimalizace kódu nesnadnější a v neposlední řadě, malý počet překladačů snižuje potenciál ke zlepšení vycházející z kompetitivnosti.

Uživatel může definovat vlastní, netriviální datové typy, zcela odstiňující vnitřní implementační logiku. Toho je dosaženo primárně konstruktory a destruktory, tedy funkcemi, které *automaticky* inicializují, resp. finalizují každou hodnotu, sekundárně pak rozhraním daným sadou metod a přetížených operátorů. Mimo klasickou sadu operátorů z C také přibyly operátory pro alokaci a dealokaci, `new` a `delete`. Ty jsou alespoň částečně přítomny i v dalších vyšších jazycích a původně byly plánovány i pro C+. Ukázalo se však, že tyto operátory nezapadají do filosofie jazyka a je vhodnější je realizovat jako knihovní funkce. Pro nativní operátory totiž (v C i C+) platí tyto fakta:

- To, že jsou elementární, tedy nevytváří potenciální neoptimálnost tím, že by šly realizovat efektivněji jiným kódem v C, což v praxi znamená, že odpovídají jedné nebo několika instrukcím strojového kódu.
- Nejsou parametrizovatelné, jejich sémantika je jednoduchá a jasně daná.
- Nemohou selhat. Toto je zvlášť významné, neboť syntakticky nemají možnost explicitně informovat o chybě[2].

Vidíme, že `new` sem nezapadá ani jedním z bodů. Alokace je sice častou operací a musí být komfortní, ale není nutné pro ni zavádět zvláštní syntaxi a zakrývat fakt, že jde o volání funkce žádající správce paměti; zároveň není problém přidávat další parametry dle potřeby.

[1] Např. přidání tzv. „range-based for“ cyklu, zjedodušující běžný úkol iterace nad kolekcí

[2] Přetížené operátory pro netriviální typy, které mohou selhat např. kvůli nedostatku paměti, pak o ní musí informovat mechanismem vyjímek.

Co se týká možností generického a metaprogramování, C++ podporuje metaprogramování systémem šablon, inspirovaný Adou. Tento systém je vysoce expresivní a typově bezpečný, navíc díky tomu, že je plně vyhodnocovaný za překladu, je i vytvářený kód vysoce optimalizovatelný. Od C++11 také podporuje lambda funkce a „`constexpr`“ funkce a výrazy, které se vyhodnocují přímo při překladu. Tato funkcionalita má zatím mnoho omezení (částečne relaxovaných ve verzi C++14), umožňuje ale řešit jisté problémy imperativně stejně jako zbytek programu, zatímco dříve k podobným účelům bývala zneužívána síla Turingovské úplnosti šablon, což vedlo k velmi zatemněnému kódu. Cenou za toto rozšíření je fakt, že prekladač C++ musí obsahovat více či méně plnohodnotný intepret.

Časy překladu jsou oproti C často pomalejší, jednak kvůli vysoké složitosti jazyka (vyžadující např. sofistikovanou syntaktickou analýzu pro rozhodnutí nejednoznačností v gramatice) a také díky vyhodnocování metakódu. [24][14][15][16]

3.3 Ostatní

Python je objektovým interpretovaným skriptovacím jazykem, jehož charakteristickým rysem je vysoká čitelnost. Toho dosahuje hlavně kvalitní a rozsáhlou standardní knihovnou, po syntaktické stránce stojí za povšimnutí odsazování bílými znaky, jednoduchá práce s n-ticemi a kompaktní syntax pro práci s výřezem (`hodnota[začátek:konec:krok]`). Stinnou stránkou je to, že pro výpočetně náročnější úlohy, u kterých neexistuje nízkoúrovňový backend, není ve srovnání s přechozími jazyky výrazně efektivní. [11]

Java a C# jsou interpretované objektové jazyky vycházející z C. Oba obsahují garbage collecting, pro C+ zaměřený na výkon jsou tak zajímavé spíše syntakticky. Oproti C jsou mnohem striktnější a tedy bezpečnější. Java je starší a v mnoha směrech omezenější, nepovolující např. neznaménkové typy a práci s ukazateli. C# se v návrhu snažil poučit z syntaktických nedostatků Javy a zavádí např. velmi kompaktní syntaxi pro častý návrhový vzor getter/setter. [8] [2]

Objective-C je další tradiční kompilovaný objektový jazyk z rodiny C. Díky jeho nižší rozšířenosti, relativně exotické syntaxi objektových rozšíření C vycházející ze Smalltalku a rozhodování s nimi spjatých problémů zasílaní zpráv za běhu oproti C++ řešícímu podobné věci staticky při překladu se pro rozširování C+ nezdál vhodný.

Haskell je klasický funkcionální jazyk a jako takový je od C velmi vzdálený, minimalismus jeho syntaxe je však silnou inspirací a zejména jeho systém typových tříd pro je velmi elegatním řešením generického programování. [6]

Jazyk D je nástupce C++, snažící se vyhnout stejným problémům. Lze ho chápat jako zjednodušení C++ s odstaněním řady problémů, které v C++ přežívají z důvodů zpětné kompatibility. Systém šablon je zjednodušen a doplněn imperativním metaprogramováním, přibyl volitelný garbage collecting a zajímavá možnost volat funkce jako metody.

Moderní jazyky vycházející z C – Go a Rust. Zaměřené na nízkoúrovňové programování, neobjektové, tedy s podobnou cílovou filosofií jako C+, ale zpětně nekompatibilní s C. Oba opravují např. C syntax deklarací, která je zejména u funkčních ukazatelů neprakticky překomplikovaná. Go se podařilo odstranit středníky a má specifickou syntax typovou inferenci, Rust je zase zajímavý přístupem, kde jsou proměnné jsou implicitně imutabilní. [5] [12]

Kapitola 4

Rozšíření

4.1 Automatizace procesu překladu

4.1.1 Motivace

Ačkoli po ukončení BP překladač ovládal kompletní gramatiku z jazyka C a všechna rozšíření v ní popisovaná, stále existovala významná překážka pro praktické nasazení – chybějící podpora pro preprocesor jazyka C.

Tento spjatý jazyk je efektivně zcela paralelní ke struktuře programu v C. Jak název napovídá, předzpracovává soubor před samotným překladem. Na obsah nahlíží jako na proud textu bez zvláštního významu, až na občasné speciální direktivy; kompletně tak ignoruje strukturu a členění obsaženého programu, což přináší řadu problémů, neboť ačkoli je na konceptuální úrovni fáze preprocessingu a následného překladu zcela oddělená, reálné překladače tyto kroky spojují.

Primární účel preprocessoru je v zásadě formátovací, slouží k dělení jedné překladové jednotky[1] do více fyzických souborů, pojmenovávání konstant nebo (obecně bloků kódu) pomocí maker a pro kondicionální inkluzi části programu. Z tohoto je patrné, že pro překlad by teoreticky neměl být vůbec nutný, v praxi lze však těžko najít kód, který by ho nepoužíval, minimálně pro import deklarací funkcí ze standardních knihoven. Naštěstí, překladače obvykle[2] poskytují možnost provést samostatný preprocessing, překladač C+ ho tak striktně vzato podporovat nemusí, za cenu komplikovanějšího vývojového cyklu C+ programů.

4.1.2 Problémy

Jednou z originálních motivací při vývoji C+ byla představa, že výstupem překladače bude C soubor zcela zachovávající původní formátování a čitelnost – pro programy využívající minimum rozšíření nebude zřejmé, že byl vytvořen počítačem a pro program v čistém C bude identický.

Tento ideál má mnoho výhod. Umožnuje nasadit C+ i v prostředí, kde je vyžadováno programování v čistém C. To může fungovat např. tak, že se program nejprve napíše v C+, což např. umožní odsazování bílými znaky a automatizuje části, které by se jinak musely řešit explicitním voláním funkcí (konstruktory/destruktory/operátory). Pak se program jednorázově převede do C formy a nadále se udržuje v té podobě.

[1] V originále „Translation unit“.

[2] Minimálně zkoumané GCC [4], Clang a MSVC.

Takto by na podobné programy šly implicitně použít i veškeré nástroje specificky určené pro C – programy pro statickou analýzu kódu, debugger, profilovací nástroje a další.

A konečně, pokud by převod do C nevedl k významné ztrátě čitelnosti, C+ by nemuselo v návrhu novějších verzí zohledňovat zpětnou kompatibilitu, protože by pro nějaký projekt do budoucna vždy šlo distribuovat či udržovat jen přeloženou verzi v C. Jak se ukázalo při průzkumu jazyků, zpětná kompatibilita je v návrhu nových rozšíření často hlavním důvodem nutícím ke kompromisům a uživatelsky neintuitivnímu chování.

Bohužel, tohoto ideálu v praxi téměr nelze dosáhnout.

Uvažujme nejprve čistý jazyk bez preprocessoru. Na program lze nahlížet jako na řetězec tokenů, potom lze metainformace o formátování, komentáře apod. v rámci lexikálního analyzátoru snadno uložit jako prefix k následujcímu tokenu.[3] Problémem je ale další zpracování na vyšších úrovních, protože překladač C+ ukládá program jako vysokoúrovňovou grafovou strukturu, kde bloky jako „příkaz if“ nebo „výraz podmínky cyklu“ jsou jen obtížně namapovatelné na původní řetezec tokenů.

I kdyby byl implementován systém pro extrakci a uložení těchto dat na vstupu a přesně inverzní postup na výstupu, ostatní operace nad programovou strukturou (jako konverze z C+ do C nebo budoucí metajazyk) by byly extrémně komplikované.

Přesto zatím není problém neřešitelný, uvažme tedy jak situaci mění preprocesor. Pokud uvažujeme, že máme infrastrukturu zmíněnou výše umožnující uložit a reprodukovat odsazení a komentáře, není problém podobně ukládat preprocesorové direktivy.

První problém jsou makra. Klasický případ užátí je definice široce používané konstanty NULL 4.1, vyznačující prázdný (nevalidní) ukazatel.

```
#define NULL ((void *)0)
```

Obrázek 4.1: Definice konstanty makrem v C preprocessoru.

V tomto případě je rozgenerování tohoto makra (tedy nahrazení jeho symbolického jména obsahem) nadbytečným krokem a snadno by s ním na syntaktické úrovni šlo pracovat jako s proměnnou, bohužel je triviální vytvořit případy, kdy toto možné není 4.2.

```
#define NO_CONDITION (1)
#define HALF_CONDITION ( function()
#define TWO 1+1
if NO_CONDITION {}
if HALF_CONDITION < 23) {}
TWO*3 == 4;
```

Obrázek 4.2: Ilustrace netransparantního použití maker v C preprocessoru.

První příklad ukazuje, jak lze s makry vytvořit kód, který bez znalosti obsahu makra není validní (uvažujeme povinné kulaté závorky v podmínce `if`u), druhý příklad má navíc nevyvážené závorky. Třetí vypadá na první pohled bezproblémově, ale díky tomu, že makra fungují jako textová substituce[4], se vyhodnotí násobení před sčítáním.

[3] Tento systém byl implementován a je užitečný pro zvýšení čitelnosti chybových hlášení.

[4] Technicky vzato v C99 je obsah makra chápán řetězec tokenů, což je důležité proto, že nekonformní preprocesory způsobovaly u maker konkatenaci po sobě jdoucích identifikátorů.

Jediné dostatečně obecné řešení, které by makra ve výstupu zachovávalo, je všechny makra kompletně rozgenerovat, provést překlad do C a následně pred výstupem prozkoumat, jestli se makro dá znovu složit.

Druhý jestě závažnější problém je v kondicionálním překladu. Pokud se budeme držet našeho ideálu a zachovat C soubor stejný jako soubor C+ až na nutné transformace, nemůžeme vyhodnocovat preprocesorové podmínky, ale musíme rozpracovat všechny varianty.

Uvažme příklad 4.3, kdy má jedna konstanta dvě různé definice na základě definovaného makra. Takovou konstantou lze pak snadno změnit programový tok jen např. výběrem jiného přetížení.

```
#ifdef TEST
  #define CONSTANT ((int)0)
#else
  #define CONSTANT ((void*)0)
#endif
```

Obrázek 4.3: Kondicionální překlad může snadno způsobit významné nelokální změny ve zbytku překladové jednotky.

V dostatečně zobecněném případě se může pro každou variantu kondicionálního překladu zcela změnit zbytek překladové jednotky a vygenerovat výsledný soubor, pokrývající všechny varianty by bylo nejen obtížné, ale byl by pravděpodobně i velmi nečitelný.

Toto také znamená, že kvůli existenci maker je nemožné překládat hlavičkové soubory samostatně, protože v extrémním případě může v každém místě, kde je vložený direktivou include, vést na odlišný kód.

Pro ilustraci, tento důsledek používání preprocessoru pro modulární programování není problematický jen pro C+. Programy v C i C++ se překládají zbytečně dlouho, protože stejný hlavičkový soubor použitý na více místech musí být zpracováván pokaždé znovu, což může trvat dlouho, zvlášť pokud obsahuje celé definice funkcí (ať už jsou `inline` nebo jde o šablony). Ačkoli hlavičkové soubory typicky obsahují direktivy zabraňující nechtěnému vícenásobnému překladu, standardní přenositelné řešení pomocí direktivy `ifndef` stále nutí alespoň k lexikální analýze celého souboru.

Řada překladačů podporuje „předkompilované“ hlavičkové soubory jako rozšíření, nicméně plnohodnotný, sofistikovanější modulární systém je rozpracován v dlouhodobějších plánech vývoje C++.

Protože takové rozšíření vyžaduje podporu v překladači C, nelze ho rozumně uvažovat jako potenciální rozšíření pro C+.

Celkovým výsledkem demonstrace tedy je, že i kdyby překladač C+ zcela podporoval preprocessing, stejně by překládal celou překladovou jednotku jako velký celek, kde by rozdělění na jednotlivé fyzické soubory a nerozgenerovaná makra pouze komplikovaly situaci. V takovém případě je výhodnější nechat preprocessing provést překladačem C, což má další výhodu v tom, že C+ prekladač nemusí znát adresáře s hlavičkovými soubory knihoven.

Je zde však jeden dosud nezmíněný problém – direktiva `pragma`, určená k možnosti rozšíření jazyka specifické pro daný překladač či dialekt. Jakožto direktiva preprocessoru se může vůči okolnímu C kódu vyskytovat prakticky kdekoli a jen sémantika daného rozšíření ho ji omezuje. Například běžně podporovaná pragma direktiva `pack` se používá u definice struktury pro ovlivnění rozmístění jednotlivých členů v ní.

Při provedení preprocessingu externě (prekladačem C) se tyto informace mohou ztratit nebo zachovat, obojí je problém. Pokud se zachová na místě, kde by ho překladač C+ nedokázal přijmout, tak selže při syntaktické analýze.

Ztráta může mít následky od minimálních (jako menší optimalizační potenciál) po kritické (binární nekompatiblita).

Naštěstí, při reálných testech s GCC se ukázalo, že se `pragma` nevyskytuje příliš často a zachované instance jsou zpravidla používány jako přídavné atributy deklarací, kde jsou zpracovatelné podobně jako atributy ostatní (detaily viz 4.1.4). Proto bylo toto externí řešení preprocessingu nakonec zvoleno a exotičtější případy užití `pragma` jsou v současné verzi považovány obecně za nepodporované.

4.1.3 Realizace

Jak bylo nastíněno výše, program (konkrétněji jeden modul/překladová jednotka) pro C+ musí při překladu projít následujícími fázemi:

1. Preprocessing – zpracování maker, inkluze všech souborů tvořící jednu překladovou jednotku do jednoho apod. Realizováno překladačem C v módu „Proveď pouze preprocessing“.

2. Překlad do C – vytvořený soubor se převede z C+ do čistého C. Provádí překladač C+.

3. Kompilace C souboru – obvyklé vytvoření objektového souboru jako při běžné kompilaci jednoho modulu v C.

Jednotlivé objektové soubory se pomocí linkeru poté spojí do výsledného spustitelného souboru, opět beze změny oproti kompilaci programů v C.

Provádění tohoto cyklu ručně je však velmi nepohodlné, byť obvyklé mechanismy automatizace překladu (jako systém *make*) pomohou. Aby však byl překlad uživatelsky srovnatelně komfortní s překladem programu v C, byl vytvořen jednoduchý sekundární program *cpmake*, zapouzdřující výše zmíněnou posloupnost a fungující tak jako frontend pro samotný překladač.

Program cpmake vyžaduje konfigurační soubor, ve kterém jsou definovány všechny statické informace (stejné pro každý překlad), tedy cesty k překladačům a jejich parametry. Zvlášť se definuje překladač pro preprocessing, C+ překladač, C překladač a linker. Díky tomuto se pak dá cpmake jednoduše ovládat z příkazové řádky, kde v základním módu stačí zadat jména jednotlivých modulů a lze přímo vytvářet spustitelný soubor popř. neslinkované objekty, stejně jako s libovolným jiným překladačem. Na příkazové řádce lze díky přepínači poslat dodatečné parametry nejen pro cpmake, ale i libovolnému článku v překladovém procesu.

Program byl napsán v C++ pro jednodušší práci s textem. Protože spouštění podprogramů s pozastavením mateřské aplikace není součástí funkcionality standardních knihoven, tato část musela být řešena pro každý operační systém zvlášť. Podporovány a otestovány byly jak POSIX systémy, tak systém Windows.

Cpmake úspěšně pokryl zadanou úlohu a teoreticky by měl být flexibilní natolik, že dokáže propojit C+ s libovolným konformním C překladačem.

4.1.4 Kompatibilita s nekonformními překladači

Bohužel, reálné testy s GCC odhalily vážné komplikace.

Překladač GCC podporuje řadu standardů, implicitně ale pracuje s vlastním rozšířením jazyka C nazývaným GNU C. Předpokladem bylo, že pokud se při překladu vynutí standard C99, výsledkem skutečně bude konformní kód, nicméně se ukázalo, že některá rozšíření v hlavičkových souborech standardních knihoven přetrvávají. Co je horší, tyto dodatečné informace jsou nezbytné k úspěšnému překladu, typickým příkladem je informace o nestandardní volací konvenci pro danou funkci.

Kromě toho, překladače mají vlastní, zabudované datové typy, které nejsou ve standardním C. V GCC jde např. o implementaci funkcí předávajících proměnlivý počet parametrů. Standard C specifikuje, že informace o parametrech je uložena v datovém typu `va_list`, jeho obsah ale definován není. V GCC je `va_list` pouhý alias na typ `__builtin_va_list`, který ale nikde definován není. Překladač C+ tak, bez specifické podpory pro GCC, selže při analýze kódu kvůli neznámemu typu.

Druhý problém očividně vyžaduje specifický konfigurační soubor, který by pro překladač C+ dodefinoval zabudované typy a funkce pro daný překladač. Řešení je překvapivě snadné – takový soubor může mít obvyklou C(+) syntaxi a chovat se jako libovolný hlavičkový soubor, stačí ho pak automaticky zanalyzovat před překladem zbytku programu (a pochopitelně vynechat při výpisu výsledku).

První problém je mnohem náročnější za předpokladu, že ho chceme vyřešit obecně. Cílem je navrhnout systém, který by dokázal nadeklarovat formát rozšíření, při syntaktické analýze je detekoval a uložil tak, aby se na výstupu objevily beze změny na stejné pozici.

Překladače pro svá rozšíření používají různé formáty, např. Clang podporuje hned pět. Prvním mechanismem je preprocesorová direktiva `pragma`, která ale není příliš oblibená, protože není součástí syntaxe jazyka – musí být na samostatném řádku, není na pohled jasné, ke kterému prvku se váže a nemůže být výsledkem rozgenerování makra[5]. Tento mechanismus kvůli externímu preprocessingu není podporován, překladače ale stejně preferují jiné, přímo rozšiřující gramatiku jazyka. Tyto rozšíření se běžně nazývají *atributy* a vážou se ke konkrétnímu prvku jazyka, který modifikují, např. definici struktury, deklarace funkcí nebo i libovolný příkaz. [1]

Jazyk C++ ve verzi z roku 2011 standardizoval mechanismus zápisu atributů v dvojitých hranatých závorkách, patrně inspirovaný podobnou syntaxí z C#. Clang tuto konstrukci povoluje i jinde. C+ tuto syntaxi v dlouhodobém výhledu také plánuje adoptovat, ovšem pro své vlastní atributy, které mají sémantiku přímo definovanou v programu, toto rozšíření pak bylo expandováno na celý metajazyk. (viz 4.7.3).

Další systém používá specifická klíčová slova, jako je např. `__fastcall`.

GCC má svá rozšíření unifikované do tvaru `__attribute__((...))`, kde uvnitř kulatých závorek je konkrétní rozšíření, MSVC podobně používá svůj `__declspec`. Vidíme, že tyto případy můžeme chápat jako parametrizovatelné klíčové slovo. [9]

Řešení v C+ je následující. Byla zavedena nová syntax pro definici atributů (4.4). Jde zadefinovat jak atribut typu klíčové slovo, tak atribut „funkční“, který je následovaný kulatými závorkami obsahující libovolnou sekvenci libovolných tokenů[6].

Všechny atributy (libovolného typu) jsou rozeznány jako unikátní typ tokenu a gramatika jazyka je rozšířena tak, aby šlo na každém místě, kde jsou podporovány, přijímat jejich

[5] V C99 toto explicitně řeší operátor `_Pragma`.

[6] S vyváženým počtem kulatých závorek.

```
#pragma attribdef jmeno_atributu1
#pragma attribdef jmeno_atributu2(...)
```

Obrázek 4.4: Formát definice atributů realizovaný standardní metodou rozšiřování jazyka – direktivou `pragma`. Protože preprocessing by definici odstranil, může se vyskytnout jen v konfiguračním souboru pro daný překladač, ten preprocesován není.

```
typedef void * __builtin_va_list;
extern void __builtin_va_start(...);
extern void __builtin_va_end(...);
long long __builtin_llabs(long long);
#pragma attrdef __attribute__(...)
#pragma attrdef __asm__ * (...)
#pragma attrdef __extension__
```

Obrázek 4.5: Ukázka pomocných definic atributů a zabudovaných datových typů nutných k přeložení jednoduchých programů využívající základní knihovny `stdio/stdlib` v GCC 4.9.2.

neomezenou posloupnost. Všechny atributy vztahující se např. k jednomu identifikátoru deklarace jsou uloženy v seznamu a při výstupu opět vypsány.

Tento systém již umožňuje úspěšný překlad programů s GCC, do budoucna by ale šlo rozšířit počet míst, kde se mohou atributy vyskytovat, přidat specifikaci omezení, na které třídy prvků jde daný atribut použít a mít volitelnou asociativitu – např. atributy stylem C++11 se k deklarátorům přimykají zleva, GCC `__attribute__` zprava.

4.1.5 Konkrétní problémy s GCC

První testovanou verzí bylo GCC 4.9.2. Jak bylo řečeno dříve, dialekt jazyka používaný překladačem se výrazně odlišuje od oficiálního C, odchylky ale lze korigovat speciálním hlavičkovým souborem s definicemi, který je konceptuálně implicitně vložen (directivou `include`) na začátek každé překladové jednotky. Možný obsah tohoto korekčního souboru je uveden v 4.5.

Ukázalo se však, že toto není jediná komplikace. Kvůli podpoře různých oficiálních i neoficiálních standardů (od C89 přes GNU C99 až po C11) už nejsou později přidaná klíčová slova jako `const`, `restrict` a `inline` v GCC rozeznávány jako skutečná klíčová slova, po preprocessingu jsou zaměněna za jejich skutečnou interní formu `__const__`, `__restrict__`, `__inline__`. V této formě jsou podporovány při libovolném zvoleném standardu, zatímco „oficiální“ formát může být v C89 módu brán jako normální identifikátor. Tato komplikace lze naštěstí snadno vyřešit tak, že se na příkazové řádce preprocessoru dodají makra, které proces obrátí, tedy např. `-D __restrict__=restrict`.

Podpora atributů je v této práci chápána jen jako problém obcházející technika, proto se mohli v gramatice z počátku vyskytovat jen pro nejmenší množinu rozumných případů použití nutných k zprovoznění kompilace, tedy navázané na deklaraci proměnné nebo datového typu. V knihovnách pro GCC 5.2.0 se však objevila deklarace takovohoto formátu, `void (__attribute__((noreturn)) ****f) (void)`, která vyžadovala ukládat atributy i pro jednotlivé úrovně indirekce komplikovaných datových typů. To nejen zkomplikovalo

vnitřní reprezentaci typu „Datový typ“, ale bylo třeba rozhodnout, jaký vliv mají atributy na identitu typu. Byla zvolena politika, kterou má GCC, tedy, že přidání `__attribute__` nevytváří nový datový typ a tedy dva strukturou odlišné datové typy mohou být pro některé účely brány jako identické. Kvůli zjednodušení realizace se také nepředpokládá, že pořadí atributů na dané úrovni hraje roli. Pokud je více atributů za sebou, je sice jejich sekvenčnost obvykle zachována, lze si ale představit obskurní promíchání s kvalifikátory, kde by bylo uchování dané permutace složité.

Dalším problémem byl příkaz inline assembleru `asm`. Na rozdíl od C++ není v C součástí jazyka, lze ho ale pokrýt jako obvyklý funkční atribut. Problémem je až to, že GCC umožňuje nečekanou syntaxi – `asm volatile (...)`. Pro řešení podobných věci byla direktiva attrdef rozšířena o speciální možnost

```
#pragma attribdef jmeno_atributu *(...)
```

kde mezi jménem atributu a otevírací závorkou může být libovolný jiný token.

GCC 5.3.0 mělo v hlavičkových souborech chybně redeklarovanou funkci `strtod`, a to tak, že v jedné z deklarací chyběl v parametru kvalifikátor `restrict`. V C toto problém není, protože funkce nelze přetěžovat, v C+ tak ale vznikaly dvě různá přetížení, což nebylo povoleno, neboť měly obě kvůli zpětné kompatibilitě přesně stejné jméno. Tento potenciálně velmi seriózní problém kompatibility s laxním C kódem také zdůraznil drobný rozdíl v přetěžování mezi C+ a C++, kde C++ zcela ignoruje nejvyšší úroveň kvalifikátorů. To je v C+ ale praktické (viz kapitola 4.3), řešením tak byl kompromis, kde jsou nejvyšší kvalifikátory ignorovány pro „unmangled“[7] variantu.

V GCC 5.4.0 se objevil nový formát klíčových slov, `__const` a `__inline`.

Konečně, v GCC 6.3.0 se v kódu na nejvyšší úrovni po preprocessingu objevil prázdný příkaz, což by podle syntaxe vůbec nemělo být možné, přestože bylo triviální tento příklad pokrýt.

Mezi verzemi GCC se interní obsah knihovenních hlaviček se tak až překvapivě lišil, s každou testovanou tedy bylo třeba řešit další drobné problémy – díky tomuto trendu je spíše nepravděpodobné, že by C+ překladač spolupracoval ad hoc s libovolným překladačem C, nebo i libovolnou verzí GCC, byť řešení této nekompatibility nebude složité.

4.2 Objektově orientovaná rozšíření

4.2.1 Motivace

Rozšířené vysokoúrovňové jazyky vycházející z rodiny C obvykle přecházejí k objektově orientovanému paradigmatu (jako Java a C#), případně jako C++ umožňují kombinovat více paradigmat s tím, že OO je dominantní. Existuje řada dobrých důvodů k tomuto jevo, objektový návrh programu je např. zvlášť vhodný pro rozsáhlé týmové projekty tím, že datové typy implicitně zapouzdřují data a omezují přístup skrz striktně definované rozhraní[8], které silně zvyšuje abstrakci. Kromě škálovatelnosti umožňuje jeho obecnost i snadné nasazení osvědčených návrhových vzorů a obecně možností softwarového inženýrství, z důvodů snadné modelovatelnosti. Na úlohy modelování a simulace je tento přístup zvlášť vhodný, což zahrnuje i např. sféru počítačových her a grafických rozhraní.

Přes toto všechno ale není OO nejvhodnější přístup v každé situaci, jak se někteří domnívají. Pro menší projekty, zvlášť vyvíjené jedním programátorem, je jeho robustnost

[7] Tedy pro speciální přetížení zpětně kompatibilní s C, které neprovádí zakódování parametrů do jména funkce („name mangling“).

[8] Formálně popsané mechanismem zasílání zpráv.

```
ListChar list;

append(list, 'A');
list.append('A');
```

Obrázek 4.6: Hypotetická ukázka dvou formátů volání funkce se sémantikou metody.

zbytečně kompikovaná a delší, přehnaně abstraktní kód vede paradoxně k nižší čitelnosti. Dále je tu fakt, že objektový výpočetní model se špatně mapuje na reálný princip, kterým pracuje procesor. To ve výsledku znamená, že za vyšší abstrakci a použití např. dynamického polymorfismu platíme nižším výkonem, což může být v některých doménách netolerovatelné (typicky operační systémy, real time aplikace, rendering).

Aby se tak C+ prakticky uplatnilo, nesnaží se konkurovat dlouhé řadě objektových jazyků, ale zůstává u strukturovaného imperativního programování, podobně jako jiné relativně mladé jazyky Go a Rust. Přesto bylo ale výhodné adoptovat některé typické rysy objektových jazyků, např. volání metod.

4.2.2 Problémy

Uvažujme datový typ `List`, obousměrně vázaný seznam. Tradičně je přidávání položek řešeno funkcemi `void append(List *, Data)` a `void prepend(List *, Data)`. Názvy jsou jasné a výstižné, problém je, že funkce stejného jména můžou existovat i pro jiné datové typy, např. `String`. Tyto situace pokrývá v C+ přetěžování, to je ale určeno pro vytváření jedné funkce, která podporuje více typů, ne pro potenciálně zcela odlišné funkce operující nad nesouvisejícími typy.

Díky implicitním přetypováním by pak snadno mohl nastat výběr nesprávné, nesouvisející varianty. Očividné řešení je mít unikátní jména, např. `list_append` a `list_prepend`, ty jsou však dlouhá. Toto manuální řešení má další problém v nejednoznačnosti netriviálních typů – jak zapsat `char *`, `enum tokenTypes` nebo `int *(*)(char)`?

Druhým problémem je formát volání funkce, který může být čitelnější, jak ukazují varianty v příkladu 4.6. Druhý formát (typická syntax volání metody v OO) na rozdíl od obecného volání funkce zdůrazňuje významnou pozici `list`u, kde `append` je metoda nad ním pracující a `('a')` její parametrizace.

Rozdíl mezi „metodou“ a normální funkcí je v prvním parametru, kterým je přenášený objekt – v C++ je nazývaný implicitní, protože se v signatuře neuvádí, jiné jazyky jako D a Python preferují explicitní zápis. Tento parametr se také v jazycích běžně označuje klíčovými slovy `this` nebo `self`, kde druhé jmenované je obvyklejší, je-li předávaný jako reference[9].

Takováto syntaxe se může objevit i v čistém C, pouze však pro ukazatel na funkci jako člen struktury. Dvojjednoznačnost významu této konstrukce je občas terčem kritiky, zvlášť proto, že ačkoli je původní význam také volání funkce, pochopitelně nedostává `self` parametr.

[9] C++ předává `this` jako ukazatel, což je ale rozhodnutí dnes vynucené jen zpětnou kompatibilitou.

4.2.3 Realizace

Jazyk D jde v podpoře metod ještě dál a umožnuje libovolnou funkci volat jako metodu s tím, že první parametr funkce musí odpovídat, tedy obě varianty v příkladu 4.6 jsou v D ekvivalentní. Tento přístup není ojedinělý, používá ho i např. vyvýjený C2.

Pro C+ byla zvolena omezenější strategie, inspirovaná tím, jak třídy v C++ vytváří vlastní jmenný prostor. Protože jedním z charakteristických rysů jazyka je to, že funkce nejsou nikdy přímo členy struktur, všechny „metody“ jsou samostatně stojící funkce. To, co jde v C+ zvané metodou, je tedy např. funkce `void List.append(List &self, Data d);`. Zápis `List.append` se podobá `List::append` z C++, kde je využito toho, že datové typy jinak operátor tečka nemají a funkce tak lze konceptuálně chápat jako člen „struktury“ tvořící daný typ.

Toto řešení může být někým považováno za méně čitelné než s :: proto, že výraz `x.f()` má odlišnou stukturu podle toho, zda je x proměnná či datový typ, ale v čistém C je běžné mít pro toto dělení zcela odlišné syntaktické stromy, viz klasický příklad `(x)*p;`[10].

Z důvodů robustnosti bylo místo výrazné komplikace gramatiky jazyka zvolen přístup definice tzv. *složeného identifikátoru*. Zatímco konvenční identifikátor v C a odvozených jazycích je tvořen jedním tokenem tvořený retězcem číslic, písmen a podtržítka, nezačínající číslem[11], gramatika idenfifikátoru v C+ lze popsat následujícími pravidly[12]:

```
identifier: simple-identifier
identifier: type-specifier '.' identifier
identifier: '(' type-name ')' '.' identifier
```

Obrázek 4.7: Rozšířená gramatika idenfifikátoru, nahrazuje výskyty původního terminálu.

Tento systém je navržem pro maximální flexibilitu, umožnující krátký zápis jako `int.print` a `string.print`, zároveň ale pokrývající i delší typy jako `(const char *).print` nebo `(enum tokentype).print`. Tento přístup také zdůrazňuje, že jde čistě o techniku systematického name manglingu a konceptuálně je tak `string.print` podobné jako dříve zmíněné `string_print` – se dvěma klíčovými výhodami. Překladač konstrukci rozumí a dokáže tedy opravit případnou chybu a především umožňuje danou funkci elegantně volat jako metodu 4.8. „Statické“ metody, tedy metody, které nemají `self` parametr, jde tímto systémem vytvářet implicitně, stejně tak jako statické členy (např. `const size_t string.max_size = 255;` je běžná deklarace proměnné).

Oproti objektovým jazykům je tu nevýhoda, že může vzniknout kolize, kdy je typ `T` struktura se členem `x` a současně existuje metoda `T.x`. Tento problém se však v praxi často neprojeví, protože při psaní metody jsou už všechny členy daného typu známy. Tento problém by se však mohl později významněji projevit u definice metody jako šablony – uvažujme např. generickou metodu `T.new` s chováním podobnou stejnojmennému operátoru v C++, pokud takto vytvoříme metodu obecně aplikovatelnou na velké množství typů, šance na kolizi jsou vysoké. Rozhodnout takovou nejednoznačnost lze v místě volání jednak volanim jako obyčejnou funkci nebo specifikací plného jména metody 4.9.

[10] Je-li x datový typ, jde o přetypování výsledku dereference. Jde-li o proměnnou, jde o závorkované násobení.

[11] Standard C99 umožňuje i podmnožinu z rozšířeného Unicode kódování.

[12] Uvažujme, že simple-identifier je běžný C identifikátor

```
ListChar list;
void ListChar.append(ListChar &self, char value); //Deklarace metody
ListChar.append(list,'A'); //Volani jako libovolna jina funkce plnou
    specifikaci jmena
list.append('A'); //Ekvivalentni predchozimu radku
```

Obrázek 4.8: Demonstrace volání metody v C+.

```
typedef struct sListString ListString;
struct sListString {
  ListString *next;
  char *value;
};

const char *ListString.value();

ListString list;
list.value; //Pristupujeme na clen 'value'
list.(ListString).value(); //Pristupujeme na 'ListString.value', coz neni
    clen struktury a~tedy se zvoli metoda
ListString.value(); //'ListString.value' je jmeno metody, tedy se
    jednoznacne voli metoda
```

Obrázek 4.9: Ukázka rezoluce výběru metody při nejednoznačnosti.

Druhý problém je v přetěžování, kdy díky možnosti pozdějšího přidání nového přetížení metody se může v různých částech kódu (obecně i v různých modulech) stejný výraz vyhodnotit na různá přetížení, tento problém je však i u přetěžování normálních funkcí a tedy se projevuje i v C++, byť striktně vzato metody typu později přidávat nelze.

Předchozí verze C+ realizovaly přetěžování operátorů speciálně pojmenovanou funkcí, tedy např. operátor sčítání pro string šel předefinovat funkcí `string__add`. Díky systému metod se toto zjednodušilo na použití speciálního, rezervovaného jména metody začínající podtržítkem, tedy `string._add`.

Ačkoli systém složených identifikátorů nedělá gramatiku nejednoznačnou, LL gramatiku bylo obtížné modifikovat pro případy bez pomocných kulatých závorek, protože je v mnoha situacích nutné odlišit výskyt datového typu jako začátek deklarace. Důvodem komplikace je to, že tokeny charakterizující datový typ vedou na zcela odlišný abstraktní syntaktický strom a tvoří tak prakticky hlavní rozvětvení na gramatiku deklarací a gramatiku výrazů resp. příkazů. Řešením jak v gramatice, tak v překladači bylo vyčlenit typové specifikátory jako speciální případ, pro který se u každého případu užití provede odpovídající větvení až po jednom tokenu, což vedlo k nárůstu pravidel.

Z praktických důvodů byl systém dále redukován a vyřazeny případy, které nedávaly praktický smysl – především není dovoleno vytvářet nový typ pojmenovaný složeným identifikátorem a tím efektivně vytvářet víceúrovňové hierarchické jmenné prostory. Kromě zjednodušení implementace bylo hlavní motivací pozorování, kde se v experimentech dlouhá, víceúrovňová jména opakovala příliš často a snížila tak celkovou čitelnost jazyka. Mimo to není dovoleno definovat složený identifikátor jako jméno parametru funkce, návěstí ani jméno `struct,union,enum`.

```
class Trida {
public:
  //Metoda tridy Trida
  void metoda(int);
  //Ukazatel na metodu, jeho typ je "void (Trida::*)(int)"
  void (Trida::*ukazatel)(int);
};

Trida t;
t.ukazatel = Trida::metoda;
t.*ukazatel(123);
```

Obrázek 4.10: Ukázka C++ - ukazatele na metodu.

```
//Ukazka C+ - ukazatel na metodu
typedef struct sTrida Trida;
struct sTrida {
  //Ukazatel na metodu, jeho typ je "void (*)(Trida &, int)"
  void (*Trida.ukazatel)(Trida &,int);
};

//Metoda tridy Trida
void Trida.metoda(Trida &self, int);

Trida t;
t.ukazatel = Trida.metoda;
t.ukazatel(123); //Efektivne zavola t.ukazatel(t,123)
```

Obrázek 4.11: Ukázka C+ - ukazatele na metodu.

U členu struktury je složený identifikátor praktický a povolený v jediném, speciálním případě: funkčního ukazatele použitého jako metoda, tedy volaného se `self` argumentem. Pro ilustraci, C++ pro ukazatele na metody zavádí poměrně komplikovanou novou syntaxi (4.10), včetně dvou nových operátorů pro volání metod – „.*“ a „->*“. Tyto nepříjemnosti plynou z implicitnosti `self` (resp. zde `this`) parametru – problém který se projevuje mimo jiné i tím, že pokud je `this const` nebo např. `const &`, musí být tato informace až za kulatými závorkami vyznačující funkční argumenty.

V C+ se pouze zavedla konvence, že pro strukturovaný typ `T` se ukazatel na funkci (metodu) `f` pojmenuje `T.f` 4.11.

4.2.4 Getter & Setter metody

Tímto názvem se označuje návrhový vzor běžný v OO, kde konceptuálně existuje hodnota či proměnná v rámci určitého typu, konkrétní implementace je ale zapouzdřena tak, že se k ní principiálně dá přistupovat jen voláním adekvátní funkce. Tato indirekce umožňuje např. kontrolovat validitu nové hodnoty, provádět úkony s zajišněním konzistence objektu, či zaznamenávání přístupu.

Pro realizaci jsou potřeba dvě přistupovací funkce (anglicky `accessors`) – `get` vracející současnou hodnotu a `set` nastavující hodnotu novou, pro přistupovanou proměnnou/člen struktury se užívá termín *vlastnost* (anglicky `property`).

```
//Soucast definice datoveho typu v~C#
...
private void resize() {/*...*/}
private int _size;
public string size {
  get {
    return _size;
  }
  set {
    resize(value);
    _size = value;
  }
}
...
```

Obrázek 4.12: Ukázka C# - definice getteru a setteru

S tímto mechanismem lze snadno namodelovat např. nezapisovatelnou proměnnou pouhou absencí funkce `set`. Vlastnost nemusí nutně jen zapouzdřovat přístup k fyzické proměnné/členu, hodnota může být načtena při přístupu či vypočtena až na požádání.

Konkrétní implementace se mezi jazyky liší. V Javě se ujala konvence, kdy se pro člen struktury pojmenovaný např. `size`, který je z venku nedostupný, přidávají veřejné metody `getSize` a `setSize`. Toto řešení je syntakticky těžkopádné, porovnejme ukázky:

```
SomeObject obj;
int i~= obj.getSize();
obj.setSize(2 * i);

SomeObject obj;
int i~= obj.size;
obj.size = 2 * i;
```

Co je horší, konvence je přidávat i triviální gettery/settery, čistě z důvodů, že do nich bude možná později nutné přidat další funkcionalitu – a předělávat kód, přistupující k vlastnosti později by bylo neúnosné.

C# toto řeší mnohem elegantněji pomocí konsturukce přímo zabudované v jazyce (viz ukázka 4.12). Co je důležitější, volání těchto metod je skryté a navenek se syntakticky vlastnost tváří jako obyčejný člen struktury, není tak problém veřejný člen kdykoli zapouzdřit do getterů/setterů.

Dodání podobné funkcionality do C+ bylo motivováno následujícím objevem mezery v jazyce. Máme-li normální funkci `int func(int x);`, pak má výraz `func(1)` typ `int` a výraz `func` typ funkce `int (int)`, resp. ukazatele na funkci – `int (*)(int)`. Metody jsou pro většinu účelů jen syntaktická nádstavba na běžné funkce, ale pokud máme typ T a metodu `int T.metoda(T&);`, pak je pro proměnnou `T var;` volání `var.metoda()` validní výraz typu `int`, ale `var.metoda` nemá definovaný význam[13]. To otevřelo možnost dodefinovat sémantiku této konstrukce následovně:

Pro proměnnou `T var;` při existenci funkce `T.vlastnost` je výraz `var.vlastnost` exkvivalentní následujícímu:

[13] V zásadě by za typ tohoto výrazu šel považovat uzávěr nad funkcí `T.metoda` s referencí na proměnnou `var`.

1. Je-li (pod)výraz tvaru `var.vlastnost = hodnota`,
   pak odpovídá volání metody `var.vlastnost(hodnota)`.

2. V opačném případě (pod)výraz odpovídá volání metody `var.vlastnost()`.

Toto velmi jednoduché rozšíření umožňuje syntaktický komfort práce s vlastnostmi, přestože jde jen o alternativní formát volání metod, které samy jsou jen alternativním formátem volání funkcí.

4.3 Reference jako alternativa ukazatelů

4.3.1 Motivace

Reference jsou konceptem přístupu ke vzdáleným hodnotám, paralelní k ukazatelům. Zatímco typ ukazatel je jednoduše adresa v paměti a dovoluje tak kromě přístupu na adresu (dereference) také provádění libovolné aritmetiky, reference jsou obvykle definovány abstraktně na základě své sémantiky – práce s referencí na typ `T` je stejná, jako bychom přímo pracovali s hodnotou typu `T`. Vidíme, že reference se chová podobně jako ukazatel, který je na každém místě použití dereferencovaný, tedy jazyk s ukazateli reference nutně nepotřebuje.

Nízkoúrovňové jazyky jako C podporují pouze ukazatele, což reflektuje jejich pozici jako „vysokoúrovňový jazyk symbolických instrukcí“, neboť instrukční sady procesorů fungují na stejném principu. Naopak vysokoúrovňové jazyky se snaží přímou práci s pamětí z důvodů bezpečnosti zcela odstínit a pracují jen s referencemi[14]. To přináší několik výhod. Oproti identickému kódu s ukazateli není potřeba provádět dereference, což je kratší a redukuje možnost chyby. Reference také nepodporují ukazatelovou aritketiku, což opět znemožňuje spjatou třídu chyb. Protože sémantika referencí diktuje, že by měly ukazovat jen a pouze na objekt daného typu (nebo být prázdné, pokud to jazyk podporuje), jazyk si může udělat lepší představu, kde všude je daná hodnota referenovaná, což je ozvlášť důležité v interpretovaných jazycích s garbage collectingem, jako je Java, C# nebo Python. V takových jazycích je práce s referencemi implicitní a častější než přímé předávání hodnot.

C++ je jeden z mála jazyků, které mají jak ukazatele, tak reference. Tento jazyk je terčem kritiky, že za svou dobu existence nabral díky zpětné kompatibilitě řadu duplikované funkcionality[15], tady to ale neplatí zcela, neboť C++ reference plní specifickou úlohu. Mají klíčová omezení – nikdy nejsou prázdné (vždy musí referencovat validní hodnotu) a po celou dobu života odkazují na stejné místo, splňují tak v C++ silně prosazovanou filosofii RAII[16].

Proměnná typu reference má povinný inicializátor, pozdější přiřazení je pak chápáno jako operátor přirazení referencované hodnoty. Reference žádným způsobem nevlastní referencovanou hodnotu, tedy doba života hodnoty je na referenci zcela nezávislá, nicméně pokud je referencovaná hodnota na zásobníku (lokální proměnná nebo funkční argument), díky RAII nemůže reference žít déle než hodnota.

Reference v C++ tedy oproti ekvivalentnímu použití ukazatelů plní dvě různé funkce. Jednak tím, že jsou validní implicitně, což ušetřuje obvyklé testy na prázdný ukazatel a má dokumentační potenciál. Tento potenciál se projevuje i v sekundárním významu, mnohem specifičtější sémantikou paramentru funkce.

[14] Každý ze zkoumaných jazyků podporuje ukazatele v nějaké formě, minimálně pro interoperabilitu s knihovnami v jiném jazyce (kompilovaném, s typicky C hlavičkovým souborem popisujícím rozhraní).

[15] Např. pole z C vs `std::array`, `malloc` vs `new` a `std::vector`, funkční ukazatele vs `std::function`.

[16] Resource Aquisition Is Initialization.

```
//Reference na typ T
T & <=> T * const
T const & <=> T const * const
//Move reference typu T
T && <=> T * const restrict
```

Obrázek 4.13: Ekvivalence vnitřní implementace referencí v C+.

```
float pi = 3.1415926; //Globalni promenna
int const &ipi = pi; //Pokud je ipi ukazatel, kde je odkazovana hodnota?
```

Obrázek 4.14: C++ kód vytvářející skrytou globální proměnnou.

Uvažujme funkci `int func(int, char &)`, kde druhý parametr jasně říká, že pracuje s proměnnou (resp. obecně L-hodnotou) typu `char`, v `const`-korektním kódu navíc můžeme předpokládat, že hodnotu modifikuje. Naproti tomu signatura funkce `int func(int, char *)` o sémantice nic neprozrazuje – jde o *hodnotu* typu ukazatel, používanou dovnitř funkce jako číselný hodnotový typ? Je to ukazatel na L-hodnotu typu char? Pokud ano, může být prázdný? Jde o *pole* hodnot typu `char`, kde je jeho velikost známá implicitně (či předávaná prvním parametrem)? Jde o textový řetězec ukončený nulovým bytem? Nebo jde o výstupní parametr, efektivně druhou návratovou hodnotu? A pokud ano, je tato hodnota volitelná, tj. může být ukazatel nulový?

Vidíme, že reference pokrývají limitované případy použití, zato ale jsou specifičtější a umožnují komfortní předávání L-hodnot do a z podfunkce. Kromě klasických L-hodnotových referencí C++ umožňuje v referencích na konstantní typ T předávat i R-hodnoty daného typu a od C++11 umožňuje také speciální přetížení pro tzv. *R-hodnotové reference*. Jde o speciální, disjunktní typ přetížení pro realizaci „move-sémantiky“, optimalizovaného přesunu hodnoty u typů s konstruktory a destruktory bez zbytečného kopírování. Protože valná většina těchto netriviálních datových typů[17] umožnuje přesun výrazně efektivněji než duplikací, je tato funkcionalita relevantní i pro C+.

4.3.2 Realizace

V jazyku C+ jsou reference definovány podobně jako v C++, jen méně abstraktně – jde o speciální podtyp konstantních ukazatelů, které jsou implicitně dereferencovány.

Taková definice umožňuje kompletní interoperatibilitu s C, to jest např. volat funkce s referencemi v parameterech. Kromě toho nezvyšuje abstrakci, programátor tak má přesnou představu o složitosti napsaného kódu a optimalizačního potenciálu.

Zároveň se tím, že jsou reference definovány jako ukazatele, mělo za cíl zdůraznit fakt, že reference má pouze vytvářet pohled na danou hodnotu, hodnota není v referenci uložena. To v C++ neplatí vždy, příklad 4.14 ukazuje situaci, kdy by se k referenci efektivně musela vytvořit další, skrytá proměnná.

V C+ referencovaná hodnota musí mít odpovídající typ. Původním záměrem při návrhu referencí bylo nedovolit předávání R-hodnot referencemi vůbec (mimo move reference) –

[17] Typy s konstruktory a destruktory, obvykle konceptuálně vlastnící nějaký zdroj jako alokovanou paměť či otevřený soubor.

```
//Predavani T hodnotou
int func(T val);
int func(T const val); //Kvalifikovana varianta
int func(T volatile val); //Jine varianty kvalifikatoru ...
int func(T const volatile val); //... ktere se navic muzou kombinovat!

//Predavani T referenci
int func(T &val);
int func(T const &val); //Kvalifikovana varianta
int func(T &&val); //"Move" reference, C+ alternativa R-hodnotovych
    referenci z~C++

//Predavani T ukazatelem
int func(T *val);
int func(T const *val); //Kvalifikovana varianta
```

Obrázek 4.15: Přehled různých variant předávající jeden parametr typu T.

zdůvodněním bylo, že případ, kdy se R-hodnota typu `T` má předávat skrz `f(T const &)` lze rovnou řešit přetížením `f(T)` bez přídavné režie indirekce. Pozdější analýzou se však ukázalo, že by to v mnoha případech vedlo k nutnosti vytvářet více přetížení, než je nutné, `T const &` tedy R-hodnoty přijímá. Implementace předávání R-hodnoty referencí je jednoduchá - v místě volání se vytvoří dočasná lokální proměnná a do reference se uloží její adresa.

4.3.3 Problémy

Přidání referencí vede na komplikovanější typový systém[18], zejména však výběr přetížení. Uvažujeme-li libovolný typ T, lze si představit velké množství funkcí předávající různým způsobem T jako jeden z parametrů 4.15, otevřenou otázkou je, která přetížení mohou nastat současně, aniž by jejich volání bylo nejednoznačné. Složitost tohoto problému roste ve funkcích s více parametry.

Pro řešení tohoto problému musíme systematicky roztřídit různé případy puužití předávání argumentů pro různé třídy datových typů. Základní myšlenkou je fakt, že i funkce s více přetíženími by mělo daný parametr využívat pro stejné využití, nelze tedy dělat přetížení, které by jedna varianta první parametr chápala jako vstupní a jiná jako výstupní.

V první řadě můžeme z dalšího uvažování vyloučit ukazatele, neboť hodnota typu `T` se na `T *` dá převést pouze explicitně. Ukazatele mají význam pro sémantiku *volitelné* hodnoty, tedy mají stejné případy použití jako srovnatelné formy referencí.

První úrovní dělení je, zda je parametr vstupní, výstupní či vstupně/výstupní – tato kategorie by měla u konkrétního parametru být stejná pro všechna přetížení.

Výstupní parametry se chovají jako alternativní návratové hodnoty. Pro srovnání, C# tento koncept modeluje klíčovým slovem `out` u paramentru, v C+/C++ jde o parametr typu `T &`, který má nedefinovanou hodnotu při vstupu do funkce a očekává se přiřazení.

Vstupně/výstupní parametry mají modifikační sémantiku a očekávají L-hodnotu. Pro srovnání, C# tento koncept modeluje klíčovým slovem `ref` u paramentru, v C+/C++ jde opět o parametr typu `T &`. Typické použití je například `self` reference u metod.

[18] V C++11 to např. vedlo na rozšíření kategorií hodnot z L-hodnot a R-hodnot na celých 5 tříd a nejednoznačnost zda `typeof(x)` pro `int &x` vede na `int` nebo `int &` si vynutila přidat mechanismus `declspec`.

Vstupní parametry jsou komplikovanější, rozlišil jsem tři různé podkategorie:

- Náhled. Pracujeme s extérní hodnotou, kterou ale nemodifikujeme. Typický příklad je metoda, která nemodifikuje mateřský objekt, nebo funkce `strlen`. Obvyklý formát parametru je `T const &`.
- Kopie. Vytváříme novou hodnotu, na extérní hodnotě nezávislou. Standardní sémantika pro předávání hodnotou v C. Obvyklý formát je `T const &` nebo `T`.
- Přesun. Vytváříme novou hodnotu, která vznikla přesunem zdroje drženého v extérní hodnotě. Má smysl jen pro typy, které dokáží řešit přesun efektivněji než vytvářením kopie, jinak jde o předchozí případ použití. Obvyklý formát je `T &&` nebo `T`.

U případů Kopie a Přesun také hraje roli, jestli novou hodnotu využíváme přímo v podfunkci, nebo je funkce pouze prostředníkem (např. přetížený operátor přiřazení). Pokud je nová hodnota využívaná přímo, nejoptimálnější řešení je vždy předávat přímo hodnotou, tedy typem `T`, neboť lokální proměnná je nutná a indirekce referencí by vedla jen ke zpomalení.

Datové typy byly v tomto kontextu děleny podle tří kritérií:

- Podle komplexnosti. Triviální typy typicky nemají přetížené operátory, rozhodně však nemají destruktor a nedrží tak zdroje, které je potřeba automaticky uvolňovat. Netriviální typy jsou doplňkem.
- Podle velikosti. Malé typy lze efektivně předávat jako argumenty funkcí, při vhodných volacích konvencích mohou být uloženy v registrech. Velikost mají obvykle stejnou nebo menší než ukazatel. Naopak velké typy jsou typicky rozsáhlejší struktury, které se předávají odkazem.
- Podle možností kopírování/přesunů. Pojem *kopírovatelný* znamená, že typ podporuje vytváření kopií, pojem *přesunutelný* to, že dokáže svůj obsah přesunout z jedné hodnoty do druhé efektivněji, než dokáže naivní řešení (vytvoření kopie v nové hodnotě a následné vyprázdnění staré). Existují 4 podtřídy:
  - Statický. Typ nelze kopírovat ani přesouvat. Typický příklad jsou typy, kde je adresa L-hodnoty klíčová část hodnoty samotné, jako `std::mutex` a `std::atomic`.
  - Kopírovatelný[19]. Typické nativní C typy, jako `int` či `char *`.
  - Přesouvatelný. Typ lze kopírovat, přesuny jsou lze ale výrazně zefektivnit. Klasický příklad jsou typy s dynamicky alokovanou pamětí jako `string` nebo `matrix`.
  - Unikátní. Tento typ lze přesouvat, ne však kopírovat. Typický příklad je typ držící unikátní zdroj – jako `std::unique_ptr`.

Možné případy použití pro datové typy jsou tedy následující.

Výstupní a vstup/výstupní případy na typech nezáleží, používá se `T &`. Pokud je cílem vytvořit instanci typu v podfunkci, používá se `T` jak při kopírování tak při přesunu. Ostatní případy:

[19] C++ má odlišnou terminologii – „kopírovatelný“ je speciální případ „přesunutelného“ typu, patrně však neexistuje praktický případ, který by (v této terminologii) vyloženě zakazoval přesuny a přitom byl kopírovatelný – terminologie v C+ tedy nic nevynechává.

- Libovolný statický. Povoluje pouze Náhled a to skrz `T const &`.
- Kopírovatelný-triviální-malý. Náhled i Kopie přímo `T` (případy disjunktní sémantikou parametru).
- Kopírovatelný-triviální-velký. Náhled `T const &`. Vidíme, že bez podpory pro R-hodnoty by bylo nutné přidat přetížení pro `T`. Kopie `T const &`, vyjímečně `T &`, pokud je vytvářením kopie ovlivněna původní hodnota.
- Kopírovatelný-netriviální. V obou případech je preferováno předávání referenci všude kde to je vhodné, proto jde o stejný případ jako předchozí.
- Přesouvatelný-triviální. Neexistuje.
- Přesouvatelný-netriviální. Náhled `T const &`, Kopie `T const &` nebo `T &` a nově Přesun `T &&`.
- Unikátní-triviální. Neexistuje.
- Unikátní-netriviální. Náhled `T const &`, Přesun `T &&`.

Jak bylo řečeno, většina případů užití se vylučuje, současně mohou v jednom parameteru nastávat jen Kopie a Přesuny, v praxi jen `T const &` a `T &&`.

Poslední dělení hodnot na místě volání bude podle toho, jestli jde o L či R-hodnotu, zda a jaké má případné kvalifikátory (zejména `const`) a zda je hodnota explicitně přesouvána. Pokud není uvedeno jinak, `T &` by měla být identická s `T`. Pouze L-hodnoty podporují `T &`, varianta `T &&` vyžaduje explicitní přesun. Oproti tomu R-hodnoty implicně `T &&` preferují před `T const &`. Výsledná tabulka je zde:

nekvalifikovaná L-hodnota	1. T & | T 2. T const & | T const
const L-hodnota	1. T const & | T const 2. T
přesunutá, nekvalifikovaná L-hodnota	1. T && | T
přesunutá, const L-hodnota	neexistuje
nekvalifikovaná R-hodnota	1. T && | T 2. T const & | T const
const R-hodnota	1. T const & | T const 2. T
přesunutá, nekvalifikovaná R-hodnota	1. T && | T
přesunutá, const L-hodnota	neexistuje

Obrázek 4.16: Priority volby přetížení (stejný řádek = identické).

Vidíme řadu symetrií a fakt, že `T &` a `T &&` se vylučují. Je zde také menší odlišnost od C/C++, které pro parametry a výstupy z funkcí automaticky odstraňuje nejvyšší kvalifikátor, lze tedy rozlišovat mezi `int &` a `int const &`, ale už ne mezi `int` a `int const`. C+ toto pro symetričnost umožňuje.

Výsledný algoritmus rezoluce přetížení je tedy modifikován následovně. Pro každý argument je zjistěno, jestli jde o L/R-hodnotu a zda je/není explicitně přesunut, podle tohoto

jsou zakázány některé varianty, viz předchozí tabulka 4.16, jinak jsou reference chápány jako referencované typy. Úrovně podobností „přesná shoda“ až „jiný `typedef`“ jsou rozšířeny na 3 podúrovně:

1. Přesně stejné kvalifikátory.
2. Více kvalifikátorů (např. nekvalifikovaný na `const` nebo `const` na `const volatile`).
3. Jiná kombinace kvalifikátorů (pokrývá zbylé případy jako při odstranění kvalifikátorů v C/C++).

4.4 Konstruktory a destruktory

4.4.1 Motivace

Konstruktory a destruktory jsou klíčovou součástí jazyka, který podporuje *netriviální* datové typy. Zatímco triviální datové typy jsou zpravidla validní v každé konfiguraci a jde o elementární typy poskytované hardwarovou architekturou nebo jejich kompozice, netriviální typy vychází z komplexnějších konceptů, jako „textový řetězec proměnlivé velikosti“, „binární strom“, „celé číslo s neomezenou přesností“, „otevřený soubor“ a podobné. To, v jaké formě jsou skutečně uloženy na disku, je tak spíše vedlejší implementační detail. Netriviální typy jsou charakteristické spíše pro objektový návrh, z pragmatického hlediska jsou ale velmi výhodné i pro typicky imperativní kód stylu C, kde rozšiřují portfolio „zabudovaných“ typů, klasickým příkladem tohoto jevu napříč jazyky je typ `string`.

Aby byla hodnota netriviálního typu vždy v logicky validním stavu, je nutné její reprezentaci na začátku jejího života z nedefinované kombinace bitů automaticky inicializovat *konstruktorem* a na konci uvolnit zdroje v *destruktoru*. Pevně svázat držení zdrojů, jako je dynamicky alokovaná paměť, síťové spojení či otevřený soubor, s životem datového typu, který nad nimi operuje, je velmi efektivní strategie, neboť zaručuje uvolnění bez toho, aby na něj musel programátor myslet – management zdrojů je hlavní motivací k existenci netriviálních typů v C+.

4.4.2 Realizace

Idea implementace konstruktoru je jednoduchá. Datový typ T, který ho využívá, definuje metodu `T._ctor`. Tato funkce je pak implicitně zavolána ihned po každé deklaraci proměnné typu T. Rozlišujeme dva typy konstruktorů – tzv. void konstruktory, které nepřijímají žádné parametry a ostatní, které je vyžadují. Druhá kategorie po programátorovi vždy vyžaduje dodatečné parametry vždy vyplnit, idea oproti normální funkci plnící stejný účel je v tom, že (pokud neexistuje i void přetížení) není možné tyto údaje nazadat.

Existují dvě možnosti, jak by se dal formát funkce `T._ctor` nadefinovat.

```
T T._ctor(/* mozne dalsi argumenty */);
void T._ctor(T &self,/* mozne dalsi argumenty */);
```

Oba formáty mají své výhody. Vracení výsledku hodnotou je snadněji optimalizovatelné, zvlášť pro typy dostatečně malé, aby šly předávat v registrech. Pokud vytvářená hodnota není L-hodnotou, nemusí tak v některých případech vůbec být v paměti. Kromě toho, tento formát odstraňuje příležitost k chybě, kde neznalý programátor čte/kontroluje `self`, který má nedefinovanou hodnotu.

Druhý formát je zase konzistentnější k obecnému tvaru metod a je efektivnější, pokud pro validní iniciální hodnotu není potřeba nastavovat všechny členy typu T.

Protože jde oba dva formáty rozlišit návratovým typem, není teoreticky problém podporovat oba dva, v překladači je v současnosti z praktických důvodů podporován pouze druhý.

Destruktor je metodou ve tvaru `T._dtor(T &self)` a jeho implementace v C je náročnější. Musí být zavolaný při každé příležistosti, kdy relevantní proměnné skončí rozsah platnosti. Místa, kde toto může v normálním toku nastat jsou následující:

- Konec bloku, tedy pravá složená závorka[20] u složeného příkazu. Ukončuje všechny lokální proměnné definované o jednu úroveň níže, tedy v těle bloku.
- Konec těla funkce. Podobná předchozímu, ukončuje navíc všechny funkční argumenty.
- Příkaz `return`, ukončující všechny lokální proměnné a funkční argumenty, vyjímkou jsou situace, kde je některá lokální proměnná/argument přímo vracena hodnotou.
- Příkaz `continue` a `break`. Ukončuje všechny proměnné definované mezi daným místem a relevantním cyklem. `break` navíc ukončí i možné proměnné definované v inicializaci cyklu, což umožňuje `for`.
- Příkaz `goto`. Chování závisí na relativní pozici cíle skoku.
  - Cíl je v kódové hierarchii pod `goto`, nebo na stejné úrovni. Žádná proměnná tedy nemohla zaniknout, je však možné, že byla přeskočena deklarace jedné nebo více nových proměnných, to by vedlo na jejich nedefinovanou hodnotu. V C++ je takový skok zakázaný, v C ovšem ne – C+ zde obětuje zpětnou kompatibilitu a také tento případ nepovoluje.
  - Cíl je v kódové hierarchii nad `goto`. Identická situace jako u příkazů `continue` a `break`.
  - Cíl není v kódové hierarchii striktně nad, ani pod `goto`. V tomto případě se musí nalézt nejbližšší úroveň (blok kódu), pro který platí, že je pod ním jak příkaz `goto`, tak cíl skoku. Pak lze skok rozložit na dvě fáze – skok nahoru na zmíněnou úroveň a skok dolů ze zmíněné úrovně na cíl. Obě fáze jsou zpracovány jak bylo popsáno dříve.

Proměnné jsou destruovány v opačném pořadí oproti tomu, ve kterém byly deklarovány.

Hodnota netriviálního typu může také vzniknout v rámci výrazu. Taková hodnota je zachycena do pomocné proměnné a destruována po jeho konci. Hodnoty netriviálních typů lze i vytvářet pomocí speciální syntaxe jako v C++, nicméně tzv. složené literály z C99 ve tvaru `((T) {*inicializace*})` pro toto záměrně nejsou využity – ty pro netriviální typy umožňují vytvořit lehkou, trivializovanou instanci, na které se konstruktory ani destruktory nevolají.

4.4.3 Problémy

Syntaxe konstruktorů v C++ je značně problematická. Volání konstruktoru jakožto funkce, tedy mechanismus vytváření R-hodnoty netriviálního typu vypadá `T()`, resp. `T(*atributy*)`. Toto řešení je elegantní a používané v celé řadě jiných jazyků, ale v C/C++ je gramaticky

[20] Pro bloky ohraničené bílými znaky jde pochopitelně o konec odsazené části.

```
T x(String());
//Vyznam 1 - x je promenna typu T s~explicitnim volanim konstruktoru, ktery
    bere parametr typu String. Hodnotou je string vytvoreny prazdnym
    konstruktorem

//Vyznam 2 - x je deklarace funkce vracejici T, berouci ukazatel na funkci
    vracejici String

//Vyznam 1 je spravny, protoze ma prednost

T x((String()));
//Nyni je spravny vyznam 2, protoze datovy typ nemuze mit nadbytecne kulate
    zavorky
```

Obrázek 4.17: Nejednoznačnost syntaxe konstruktorů v C++.

nejednoznačné, neboť např. `String()` může být vyložen jako „konstruktor typu `String`, výsledek je hodnota typu `String`“ nebo jako „funkce, která nebere žádné argumenty a vrací `String`, výsledek je datový typ“. Tento problém byl namísto změny syntaxe „vyřešen“ tak, že oba výklady jsou platné a při syntaktické analýze se tak musí vytvářet více syntakticých stromů pro různé významy s tím, že ve většině situací nakonec zůstane jen jeden. Ve vyjímečných případech, jako je 4.17, se kvůli zpětné kompatibilitě preferuje výklad z C. Pro explicitní konstruktor u proměnné C++ používa syntaxi `T jmeno_promenne(*atributy*);`, která trpí stejnými problémy.

Pro C+ tak musela být navržena syntaxe, která nebude výrazně komplikovanější, zato však jednoznačná. Byl využit operátor tečky, jako logické pokračování syntaxe metod. Pro použití explicitního konstruktoru v deklaraci je to tvar `T jmeno_promenne.(*atributy*);`, pro R-hodnotu tvar `T.{*atributy*}`.

Zkoumání typického využívání konstruktorů s přídavnými parametry ukázalo, že jsou dvě různá využití. Tím prvním je nastavení klíčových parametrů, která nejdou nastavit staticky, například u matice se v konstruktoru zvolí počet řádků a sloupců. Druhým případem je „naplnění“ hodnoty daty, například specifikace počátečních hodnot kontejneru. Toto není ideální, neboť v řadě případů by datový typ mohl podporovat obojí, což může vést ke zmatení, uvažme následující teoretický příklad.

Mějme typ `vector`, dynamicky alokované pole generického typu T. Uvažujme následující konstruktory ():

- `vector._ctor(vector &self)` – void konstruktor, vytvoří prázdnou instanci
- `vector._ctor(vector &self,int size)` – varianta, která předalokuje pole, které má `size` členů
- `vector._ctor(vector &self,...)` – varianta, která přijímá hodnoty typu T a naplní jimi `vector`.

V tomto případě by se takový konstruktor pro `T = char *` dal volat takto:

```
vector x; //Prazdny vektor
vector x.(10); //Predalokace 10 prvku
vector x.("Hello","world"); //Naplneni obsahem
```

Ale pokud bychom volili `T = int`, nastane problém v nejednoznačnosti:

```
vector x; //Prazdny vektor
vector x.(10); //Chceme predalokovat 10 prvku nebo vyplnit 1 prvek?
vector x.(1,2); //Naplneni obsahem
```

Sekundární problém je to, že s přidáním konstruktorů existují dvě syntaxe pro inicializaci, `T x.(...)` a `T x = {...}`. Jaký význam by pak měla syntaxe `T x.(...) = {...}`?

C++ tento problém dat vs. parametrů v konstruktorech ve verzi 2011 také řešilo a přišli s pseudonativním datovým typem `initializer_list` reprezentující homogenní pole zapsané syntaxí `{ ... }` a s ním spjatým novým formátem konstruktorů `T{...}` resp. `T var{...}`, který ho preferuje před jinými typy konstruktorů.

Toto je však jen částečné řešení vhodné k plnění kontejnerů, pro C+ byl navržen odlišný přístup, kde se tyto dvě úlohy rozdělí do dvou nezávislých funkcí, *konstruktoru* a *inicializátoru*. Cílem takto pojatého konstruktoru je dostat hodnotu do validního, ale „prázdného“ stavu, zatímco inicializátor, pokud je definován (jde opět o volání metody, tentokrát ve tvaru `T._init`), provádí plnění hodnotami. S tímto rozdělením lze dosáhnout zajímavých vlastností, jako možnost reinicializovat proměnnou; dále pak fakt, že (void) konstruktor je povětšinou elementární a často vůbec nemůže selhat[21]. Zatímco konstruktor je implicitní a obvykle skrytý, inicializátory jsou z principu explicitní. Inicializátor, který nebere žádné parametry, je pak funkcí, provádějící vyprázdnění.

Možné případy užití pro typ implementující konstruktor i inicializátor tak mohou vypadat jako v příkladě 4.18, který mj. ukazuje i syntaxi pro nezmíněný případ explicitního konstruktoru a inicializátoru u R-hodnoty.[22]

Je nutno zmínit, že takto navržený systém favorizuje praktiku, že netriviální typy držící zdroje mají „prázdný“ stav. V C++ je naopak častý idiom RAII, kde jsou zdroje zabírány v konstruktoru a často pak žádný prázdný stav ani nemusí existovat. To má podobnou výhodu jako reference (vůči ukazatelům) – není nutné při používání provádět test na prázdnost.

Takto definované typy však mohou být v některých situacích nepraktické, například nejde jednoduše a efektivně řešit následující případ: Uvažujme netriviální datový typ T, naším záměrem je proiterovat jistou kolekci, nalézt konkrétní záznam a teprve na jeho základě proměnnou inicializovat.

```
T promenna; //Zde je deklarace promenne
for (...) { //Iterace v~kolekci
  if (...) { //Nalezeni hledaneho elementu
    promenna = {element.x,element.y};
  }
}
//Zde je promenna vyuzivana
promenna.nejakePouziti();
```

Pokud by `T` byl striktní a měl jediný konstruktor přijímající `element.x, element.y` z příkladu, kód by musel být refaktorován, patrně méně efektivně.

Konstruktory a destruktory mají také jistá omezení. Zmíněný algoritmus umísťování destruktorů uvažuje jen klasický, strukturovaný tok kódu, destruktory nemohou být volány při neočekávaných situacích, jako je např.

1. Dlouhý skok, jaký realizuje např. standardní knihovní funkce `longjmp`.

[21] Pro čitelnost je jednoznačně výhodnější, když jsou programátorovi skryté implicitní funkce bezpečné.

[22] V současné verzi překladače z časových důvodů nebyly netriviální inicializátory implementovány, tato část tak popisuje ryze směřování jazyka v tomto směru a diskutuje důvody na pozadí.

```
//Prazdny vektor
//  vola 'void vectorInt._ctor(vectorInt &)'
vectorInt vec1;
//Prazdny vektor rezervujici 32 mist
//  vola 'void vectorInt._ctor(vectorInt &,int)'
vectorInt vec2.(32);
//Vektor naplneny hodnotami
//  vola 'void vectorInt._ctor(vectorInt &)' nasledovany
//  'void vectorInt._init(vectorInt &,int array[],int size)')
vectorInt vec3 = {1,2,3,4,5};
//Vektor naplneny hodnotami
//  vola 'void vectorInt._ctor(vectorInt &,int)' nasledovany
//  'void vectorInt._init(vectorInt &,int array[],int size)')
vectorInt vec4.(32) = {1,2,3,4,5};

//R-hodnotovy konstruktor prazdneho vektoru
//  vola 'void vectorInt._ctor(vectorInt &)'
vectorInt.{};
//R-hodnotovy konstruktor vektoru s~hodnotami
vectorInt.{1,2,3,4,5};
//  vola 'void vectorInt._ctor(vectorInt &)' nasledovany
//  'void vectorInt._init(vectorInt &,int array[],int size)')
//R-hodnotovy konstruktor vektoru s~hodnotami a~explicitnim konstruktorem
vectorInt.{1,2,3,4,5};
//  vola 'void vectorInt._ctor(vectorInt &,int)' nasledovany
//  'void vectorInt._init(vectorInt &,int array[],int size)')
vectorInt.(32){1,2,3,4,5};

//Reinicializace
vec1 = {1,2,3,4,5};
//Vyprazdneni
vec1 = {};
```

Obrázek 4.18: Ukázka rozdělění úloh mezi konstruktor a inicializátor.

2. Funkce, které se nevrací na místo volání. Ty sice mohou být anotovány běžně podporovaným atributem `noreturn`, není to však nutné. Příkladem je knihovní funkce `abort`.

3. Dynamické `goto`, jak je podporované v GCC.

Do této kategorie by zdánlivě šlo zařadit i vyjímky, kdyby je jazyk podporoval, neboť ty také nepředvídatelným způsobem mění tok kódu. Pro srovnání, v C++ je s voláním destruktorů při vyjímkách vynaložena značná režie, nebo je nutné využít architekturně specifické triky, které nejde vyjádřit v přenosném C kódu.

Konstruktory a destruktory globálních proměnných by musely být provedeny před, resp. po libovolném jiném kódu. Kromě toho, situace se může dále komplikovat, pokud se při inicializaci používají hodnoty globálních proměnných z jiných modulů. Bez podpory linkeru je toto obtížně dosažitelné a tak jsou globální proměnné netriviálních typů nepodporovány, resp. trivializovány.

Překladač také podporuje trivializaci konkrétní proměnné pomocí speciálních atributů `_builtin_nodtor`, resp. `_builtin_pod`. Tuto funkčnost bylo nutné dodat z interních důvodů pro pomocné proměnné vniklé při konverzi z C+ na C a nebyl důvod ji nepovolit i pro konvenční kód.

Pole hodnot netriviálního typu funguje tak, že se v cyklu zavolá konstruktor na každkou hodnotu, destruktory jsou také v cyklu, ale v opačném pořadí. Pro struktury obsahující netriviální typy by principiálně šlo generovat kompozitní konstruktory a destruktory automaticky, v současné verzi je ale tento úkol přenechán uživateli, což ale na druhou stranu přináší flexibilitu.

4.5 Efektivní přesouvání

4.5.1 Motivace

V minulé části byly představeny typy, které vyžadují volání funkcí při vzniku i zániku, což je prováděno automaticky a rigidně. Tento přístup plní zadaný účel, ale často vytváří neoptimální kód vůči manuální správě paměti. Uvažme klasickou situaci, kde destruktor pouze uvolňuje zdroj, pokud není prázdný – pokud je krátce před destrukcí proměnná explicitně vyprázdněna programátorem voláním vhodné metody, toto volání je zbytečné. Cena jednoho nadbytečného volání funkce (potenciálně i `inline`) nemusí být závratná, ale pro celé pole takových hodnot rychle vzrůstá.

Co je ale mnohem významější, je nutnost jazyka podporovat a preferovat přesouvání zdrojů. Myšlenka je taková, že pokud je nějaký zdroj unikátně vlastněn nějakou proměnnou (tedy proměnná nemá jen počítanou referenci na sdílený zdroj), při konvenčním přiřazení do jiné proměnné je nutné zdroj zduplikovat, což je potenciálně velmi drahá operace. Pokud první proměnná po přiřazení přestává existovat, je kopie zbytečná a originál je mazán v destruktoru.

Naším záměrem tak je navrhnout strategii, jak s datovými typy, kde mají přesuny logický smysl, efektivně pracovat a přiblížit se, či dosáhnout efektivitou ruční správy zdrojů.

C++11 si tohoho bylo vědomo a přišlo s řešením ve formě *R-hodnotových referencí*[23]. Podrobnosti jsou rozebírány v kapitole 4.3, ale myšlenkou je, že tyto reference umožňují podobně jako konstantní L-hodnotové reference přijímat jak L-hodnoty tak R-hodnoty a lze

[23] R-hodnotová reference na typ T se zapisuje T &&.

pro ně definovat vlastní přetížení. Důležitý rozdíl je ale v tom, že díky tomu, že referencují R-hodnoty, lze jejich obsah beztrestně přesunout, mají v terminologii C++ tzv. *move-sémantiku*. R-hodnoty se na R-hodnotové reference konvertují implicitně, pro L-hodnoty je nutná konverze šablonovou funkcí `std::move`.

Klasické použití R-hodnotových referencí pro relevantní netriviální typy vypadá tak, že kromě konstruktoru, destruktoru a operátoru přiřazení (kopírovacího) přidává „přesouvací (*move*) konstruktor“ a „přesouvací operátor přiřazení“. Samotná obvyklá implementace přesunu z A do B stejného typu vypadá tak, že se provede výměna obsahu A a B. To zaručí, že je předchozí obsah B správně zdestruován destruktorem A.

Poznámka. C++ syntaxi R-hodnotových referencí využívá i k realizaci tzv. dokonalého předávání[24], tedy schopnost předat skrz funkci parametr do podfunkce zcela beze změny, tj. zejména zachovávající fakt, zda jde o L-hodnotu či R-hodnotu a kvalifikátory. Toho ale není dosaženo nějakou inheretní logikou R-hodnotových referencí, ale speciální vyjímkou v definici sémantiky `T &&` pro šablony, prakticky jsou tak řešeny dva nesouvisející problémy jednou syntaxí. C+ v této chvíli forwarding neřeší.

4.5.2 Realizace

C+ se inspiruje řešením z C++ s jistými obměnami. Ty povolují agresivnější optimalizace než C++, těžící z toho, že v C+ jsou netriviální typy zavedeny striktně jen a pouze pro mechanismus komfortnějšího držení a uvolňování zdrojů, C+ si tak mohlo nadefinovat sémantiku takovým způsobem, že minimalizuje redundantní volání konstruktorů a destruktorů s využitím tohoto předpokladu[25]. Zavádí *move reference*, syntakticky odpovídající R-hodnotovým referencím, ale úžeji zaměřenou sémantikou. Jaké jsou rozdíly z hlediska přetěžování už bylo zmíněno v 4.3, mohou se vyskytovat pouze jako parametry funkcí.

Předávání R-hodnotovými referencemi v C++ negarantuje žádnou sémantiku, což limituje potenciál k optimalizaci. Move reference oproti tomu říkají následující:

Pokud je L-hodnota i R-hodnota předávána skrz move referenci do funkce, je tato hodnota po vyhodnocení funkce považována za destruovanou.

Pro R-hodnoty to znamená, že na ně nemusí (ani nesmí) být po konci výrazu volán destruktor, pro L-hodnoty, u kterých lze jednoznačně určit, že už dále nebudou použity, to platí také[26]. V opačném případě se na takovou L-hodnotu po zavolání funkce zavolá void konstruktor. Tento přístup by měl ve většině případů vést k menšímu počtu volaných funkcí.

S touto definicí je typická implementace přesunu z A do B stejného typu jednodušší – nejprve se B vyprázdní (je-li to nutné) a pak se obsah A přesune (přiřadí) do B. Protože je A po operaci považováno za destruované, není nutné jeho obsah vůbec měnit!

Oproti C++ je také preferováno provádět přesuny explicitně, za tímto účelem je zaveden nový pseudooperátor přesunu `:=`. Obvyklá forma je `a := b;`, tedy „b se přesouvá do a“. Někdy je ale nutné přesouvat do argumentu funkce, zde (a jen zde) jde použít speciální unární forma `f(:=b)`. Sémantika pseudooperátoru je pouze taková, že je argument považován za explicitně přesunutý a jako takový volí jiné přetížení (opět viz 4.16). Lze si snadno

[24] V originále „perfect forwarding“.

[25] Optimalizace v C++ se řídí tzv. „as-if“ pravidlem – je povoleno kód libovolně přeskládávat a předpočítávat, pokud je výsledek co do výsledku a vedlejších efektů ekvivalentní napsanému kódu.

[26] Tato detekce je výpočetně složitá – důležité při ní je, aby nemohla nastat situace, kdy by odstranění destruktoru na obvyklém místě vedlo k tomu, že se průchodem skrz jiným programový tok stejná, ale nepřesunutá proměnná nedestruovala.

domyslet, že syntaxe `a := b;` je pouhá syntaktická zkratka na `a._assign(:=b)`, přetížený operátor přiřazení, neboť přesuny v C+ jsou smysluplné pouze pro netriviální typy.

C+ také garantuje široce podporovanou optimalizaci z C++, kde se návratová hodnota funkce při inicializaci chová stejně jako triviální, tedy že se v následujícím příkazu nevolají žádné konstruktory ani destruktory.

```
// Necht je deklarovano 'string funkce();'
string promenna = funkce();
```

4.6 Zjednodušení syntaxe pro nekonečný cyklus

Jedno z drobnějších rozšíření, které bylo v rámci předchozí verzi jazyka realizováno, byl zjednodušení zápisu nekonečného cyklu. Pro tuto velmi častou konstrukci neexistuje v rodině C žádná dedikovaná zkratka, tedy je obvykle modelována buď jako `while (x) ...` nebo jako `for (;;) ...`, kde x je výraz vyhodnotitelný při překladu odpovídající `true`, obvykle tedy `1` nebo `true`.

Oba přístupy nejsou ideální. Výhoda `for`u je, že prázdná podmínka implicitně uspívá a není ji tedy nutné vyplňovat, přesto je tato konstrukce pro tento účel přehnaně komplikovaná. To `while` je jednodušší, přesto je z konceptualního hlediska neelegantní nutit programátora psát podmínku cyklu, kde o se o skutečném záměru programátora překladač dozví až během optimalizace.

Původní návrh byl umožnit prázdnou podmínku cyklu `while`, tedy cyklus by pak byl tvaru `while () ...`. Tato redundance v prázdných kulatých závorkách se zdála nezbytnou[27], ukázalo se ale, že existuje ještě minimalističtější, syntakticky jednoznačná možnost – umožnit složený příkaz ihned po klíčovém slovu `while`.

```
//Nekonecny cyklus zapsany klasicky
while {
  int ch = getchar();
  if (ch == EOF)
    break;
  process(ch);
}

//Nekonecny cyklus s~odsazenim bilymi znaky
while:
  int ch = getchar();
  if ch == EOF: break;
  process(ch);
```

Obrázek 4.19: Ukázka kompaktnější syntaxe nekonečných cyklů.

Tento přístup má výhodu, že nezavádí nové klíčové slovo, tedy nekomplikuje zbytečně zpětnou kompatibilitu. Lze namítnout, že pro neznalého prográtora může být takovýto zápis nečekaný a neintuitivní, ale nemá žádnou jinou, nesprávnou interpretaci. Konečně, podobné rozšíření adoptoval i jazyk Go pro cyklus `for`, existuje tak predecent z praxe.

[27] Viz předchozí průzkum možnosti definovat kulaté závorky v podmínkách volitelné zmíněný v 2.2.

4.7 Generické programování

4.7.1 Rozbor používaných přístupů ke generickému programování

Základní úlohou generického programování je zabránit zbytečné duplikaci kódu a s nimi spojeným potenciálem k chybám nekonzistencí a složitější údržbou. Umožňuje psát čitelnější a kratší programy, často s možností znovupoužitelnosti určitých elementů.

Konkrétní mechanismy, kterými se tyto cíle dosahují, se mezi jazyky liší, nejtypičtějším je ale typová generičnost, kdy je daná funkce volatelná s libovolným typem parametru, případně s omezenou množinou na základě podporované množiny operací. Generické mohou být i datové typy, lze tak vytvářet typy vyššího řádu, jako např. seznam nespecifikovaného typu T.

První přístup ke generikám je na principu *šablon*, významným představitelem této kategoie je C++, ale i D či Ada. V takových jazycích je napsaná generická funkce pouhým předpisem, podle kterého se při překladu vygenerují požadované konktrétní instance (proces se nazývá *instanciace*). Velkou výhodou tohoto přístupu je, že je stejně dobře optimalizovatelný jako ručně napsaný kód odpovídající každé instanci zvlášť. Stinnou stránkou je pak prostorová složitost, tedy množství výsledného generovaného kódu.

Jazyky jako Java a C# používají syntaxi *rozhraní*. Rozhraní specifikuje kontrakt, který musí být explicitně splněn datovým typem, který rozhraní implementuje. Tento kontrakt je obvykle ve formě požadovaných metod a na syntaktické úrovni je podobný definici třídy. V programu pak v proměnné typu rozhraní je možné držet referenci na libovolnou třídu toto rozhraní implementující a dosáhnout tak generičnosti. Implementace tohoto chování se v detailech liší[28], klíčové rysy jsou ale řešení za běhu programu (vyžadující existenci informací o typech za běhu) a relativně malý (oproti šablonám) počet instancí. Stinnou stránkou je tak časová složitost, tedy slabší možnost optimalizace podobného kódu, byť zkoumané jazyky užívající tento princip jsou interpretované a režie oproti negenerickému kódu je tak často zanedbatelná. [20]

Poznámka. Abstraktní třídy a dynamický polymorfismus dokáží pokrýt podobnou funkcionalitu i v C++, pro aplikaci v C+ ale tento mechanismus vhodný není, neboť jazyk nepodporuje dědičnost. Pro realizaci generického kódu efektivním způsobem tak byl zvolen systém na principu šablon.

4.7.2 Kritika šablon v C++

Šablony v C++ byly původně navrženy, aby umořnili robustně vyřešit problém typově generických funkcí a datových typů, jak ukazuje příklad 4.20. Tento systém je však natolik výpočetně silný, že umožňuje vyjádřit prakticky libovolný výpočet, je turingovsky úplný. Vzhledem k tomu, že je celý řešen při překladu a je typově bezpečný, stal se velmi atraktivním prostředkem pro autory generických knihoven. Zde se často silně využívá koncept SFINAE[29], tedy pravidlo, že neúspěšná instanciace není nutně chybou, z množiny přetížení však jedna uspět musí.

S tímto mechanismem se z šablon efektivně stává druhý, deklarativní funkcionální jazyk, jak je vidět v příkladech 4.21 a 4.22. Tento přístup silně koliduje s konvenčním kódem, pokud bychom např. chtěli klasickou imperativní funkci s cykly vyhodnotit už při překladu

[28] Java podporuje generika jen pro objektové typy a vytváří jedinou implementaci pro všechny (mechanismus vymazání informace o typu) s odpovídajícími přetypováními v kódu generika využívající. C# reifikuje jednotlivé instance za běhu, zvlášť pro každý primitivní typ a jednotně pro typy objektové.

[29] Substitution Failure Is Not An Error.

```
struct ListInt {
  ListInt *next;
  int data;
  void push_back(int data) {/*...*/}
  int pop_back() {/*...*/}
};
```

```
template <typename T>
struct List {
  List<T> *next;
  T data;
  void push_back(T data) {/*...*/}
  T pop_back() {/*...*/}
};
```

Obrázek 4.20: Jednoduchý jednosměrně vázaný seznam v konktrétní a generické variantě v C++.

```
faktorial 0 = 1
faktorial n = n * faktorial(n-1)
```

Haskell

```
template<int n>
struct faktorial {
  static const int value = n * faktorial<n-1>::value;
};

template<>
struct faktorial<0> {
  static const int value = 1;
};
```

C++

Obrázek 4.21: Srovnání syntaxe rekurzivního výpočtu faktoriálu pomocí C++ šablon a Haskellu.

přepsáním do šablonového kódu, šlo by to jen obtížně. Dále vidíme, že funkčnost i jednoduchého kódu realizujícího faktoriál je silně zatemněná – běžnými praktikami je deklarovat zcela zbytečné pomocné struktury, nové výčtové typy pouze k předávání hodnot, i deklarovat neexistující funkce.

Tento přístup je z pohledu návrháře jazyka nešťastný. Nejen, že je takový kód obtížně čitelný pro programátora, sémantika je skrytá i před překladačem, což se projevuje zejména velmi neinformativními chybovými hlášeními. A to nejen při vytváření kódu šablony, ale zejména až při jejím nesprávném použití – kvůli instanciaci na požádání při překladu se efektivně ignoruje idea zapouzdření, špatný vstup může způsobovat na první pohled nesouvisející chybu v těle funkce, jejíž existence by uživateli měla být skryta, a problémy se jen zhoršují s používáním dalších šablon v šablonách.

Jiné jazyky (Java, C#, Python) se podobným problémům vyhýbají tím, že jejich možnosti generik jsou výrazně omezenější v expresivnosti, uživatelsky jsou však z hlediska čitelnosti chyb mnohem přívětivější. Příkladem úspěšného řešení jsou třeba typové třídy Haskellu[30], kde jsou funkce implicitně typově generické s tím, že přijímané typy musí být přijímány všemi podfunkcemi (operátory) dané funkce. Každé funkci však lze napsat explicitní signaturu, která může funkci omezovat na konkrétní typy, nebo na instanci nějaké typové třídy (nebo více tříd), kde třída specifikuje množinu funkcí, které typ podporuje. Výsledkem volání se špatným typem tak je nejen relevantnější chyboé hlášení, ale uživatel se i dozví, jakou typovou třídu dodaný typ nesplňuje.

C++ samo si je problémů dobře vědomo, novější a plánované verze tak přináší řadu novinek zlepšující situaci. Hlavním bodem jsou *koncepty*, jejichž cílem je definovat sadu omezení datového typu. Díky tomu se na datový typ v šablony dá navázat jen typ splňující

[30]Nesouvisí s třídami z kontextu OOP, mnohem blíže mají k principu rozhraní.

```
//Tato trida zjistuje, zda je parametrizujici typ ukazatel
//Vysledek je ulozen v~'is_pointer<T>::value'
template <typename T>
struct is_pointer {
  //Bezny ukazatel
  template <typename U>
  static char is_ptr(U *);

  //Ukazatel na funkci
  template <typename X, typename Y>
  static char is_ptr(Y X::*);

  //Ukazatel na funkci
  template <typename U>
  static char is_ptr(U (*)());

  //'else' resp. 'default' pripad
  static double is_ptr(...);
    //Zneuziva faktu, ze double nema stejnou velikost jako char

  static T t;
  //Samotny test, vyuzivajici pretezovani is_ptr
  enum { value = sizeof(is_ptr(t)) == sizeof(char) };
};
```

Obrázek 4.22: Využití SFINAE v C++ pro rozhodování informace o struktuře typu. Příklad ilustruje silné zatemnění sémantiky operace typické pro intenzivní programování v tomto stylu.

dané požadavky, což má jednak dokumentační potenciál (pro čtenáře i chybová hlášení), odstraňuje řadu chybných instantiací a umožňuje i jemnější dělení případů s potenciálem k optimalizaci. Koncepty samy mají formu logických predikátů, kvůli celkové komplexnosti problému je jejich začlenení do standardu stále odkládáno[31]. Jazyk D, který z C++ vychází, šablony omezuje pomocí funkcionality `static if`, resp. „*constraints*“; v zásadě lze instanciaci dělit podle podmínky obsahující libovolný výraz vyhodnotitelný při překladu.

I C++ mimo klasických šablon postupně (od C++11) začíná povolovat vyhodnotitelné funkce a výrazy za překladu, označují se kvalifikátorem `constexpr`[32]. Pro C++17 je plánovaná i zmíněná konstukce `static if`. [16]

Mimo zmíněné problémy mají šablony také problém syntaktický – použití úhlových závorek. Obvyklým významem jsou operátory „menší než“ resp „větší než“, přidává to tak do jazyka další nejednoznačnost, kdy má výraz `a < b > c` zcela odlišné syntaktické stromy na základě `a`. Nevhodnost volby se projevila i na lexikální úrovni, kde zanořené závorky tvaru `a<b<c»` končí jedním tokenem `»` namísto očekávaných dvou `>`[33].

4.7.3 Idea řešení generik v C+

Poznámka. Tato kapitola popisuje *celkovou* dlouhodobou strategii pro generické programování v C+, to však přesahuje rozsah práce – z časových důvodů však byla realizována jen základní funkcionalita, parametrizovatelné typy a funkce. Přesto je tato část důležitá pro zdůvodnění navržených rozšíření.

Při návrhu realizace generického prigramování pro C+ byly zohledněny fakta z předcházející části s tím, že byly kladeny následující nároky:

1. Plně vyřešený při překladu, nevytvářející žádné dodatečné požadavky za běhu.
2. Čitelný pro uživatele i překladač, umožňující efektivní chybová hlášení.
3. Umožnit typovou generičnost s adekvátním omezením typů (jako koncepty).
4. Umožňující metaprogramování alespoň na úrovni kódu vyhodnotitelného za překladu (jako `constexpr`) a introspekce datových typů.
5. Výrazně jednodušší než současné (a plánované) techniky v C++.

Prvním krokem je vybavit překladač interpretem jazyka, který je minimálně schopný vyhodnotit libovolnou funkci pro kterou platí, že je u ní k dispozici definice její i podfunkcí, tedy ekvivalent `constexpr`. Jak bylo zmíněno, C++ tímto směrem už směřuje. S přidáním vhodné syntaxe pak lze realizovat jak `static if`, tak obecně vyhodnocovat volání funkcí při překladu[34].

Druhým krokem je v rámci kódu prováděného při překladu rozšířit zabudované typy o podporu *metatypů*, zejména `Datatype`. Tento zapouzdřený typ reprezentuje jeden datový typ a poskytuje adekvátní rozhraní nativními metodami k jeho introspekci a modifikaci. S touto funkcionalitou lze metakód jako zmíněný `is_pointer` z příkladu 4.22 řešit jako konvenčně napsanou imperativní funkci se signaturou `bool is_pointer(Datatype T);`.

[31] Původní návrh se týkal standardu C++11, ani v současnosti nedokončený C++17 je ale neobsahuje. Existuje však omezenější, experimentální verze podporovaná GCC. [10]

[32] Tedy např. problém výpočtu faktorialu 4.21 je od C++11 možné vyhodnotit při překladu i imperativním kódem.

[33] Tento problém je vyřešen jako explicitní vyjímka ve standardu od C++11.

[34] Pochopitelně za předpokladu, že funkce nemají vedlejší efekty.

S vhodnými rozšířeními syntaxe není problém tento mechanismus využít pro mnoho účelů.

Lze snadno realizovat omezení datových typů jako slibují koncepty, kde by se např. koncept Přesunutelný modeloval jako funkce `bool Moveable(Datatype T)`, kde se v těle funkce otestuje, že typ má dané přetižené operátory. K jednotlivým generických funkcím by se pak dodal výraz, který by musel být vyhodnocený na `true` pro úspěšnou instanciaci.

Teoreticky není při vhodném oddělení problém kombinovat části kódu vyhodnocované při překladu s normálním kódem, lze tak např. zavolat metafunkci, která vrací datový typ a ten se použije jako typ u reálné proměnné.

Třetím krokem je mechanismus šablon, ovšem ve významně omezené variantě, neboť jde stále ve většině jednoduchých případů o vhodný přístup k typově generickým funkcím. Funkce je parametrizovatelná pouze datovými typy, ne konkrétními hodnotami. O vhodnosti použití rozhoduje pouze signatura funkce, je-li třeba sofistikovanější omezení vstupu, použijí se metafunkce predikátového tvaru suplující koncepty. Instanciace přijaté funkce by pak už v korektně napsaném kódu neměla selhat, tedy není využito SFINAE.

Konečně, s těmito výrazovými prostředky jde dosáhnout i vyšší úrovně metaprogramování, která se ve statických, kompilovaných jazycích neobjevuje. Je možné umožnit metafunkcím modifikovat překládaný program, ne nepodobně systému, jakým pracuje konverze z C+ do C, šlo by tak v metakódu rozšiřovat možnosti jazyka a de facto „skriptovat“ překladač. Stejně jako je `Datatype` zapouzdřenou referencí na instanci v interní reprezentaci stejného konceptu uvnitř překladače, lze si představit i metatypy jako `Identifier`, `Literal`, `Expression` a `Statement`; takto lze reprezentovat, procházet a (s rozumnými omezeními) i modifikovat kompletní překládaný program – vhodnou alegorií je zde princip modifikace DOM[35] pro jazyk HTML.

Modifikace kódu by pak šla na syntaktické úrovni realizovat identicky jako atributy, pouze by takový „atribut“ byl funkcí v metakódu, která by dostala referenci na příkaz, ke kterému se váže.

Uveďme si jeden konkrétní příklad využití takového metakódu a ilustrující jeho činnost. Nechť existuje metafunkce `void precondition(Statement &self,Expression condition);`, která by šla použít jako atribut definice funkce, řeknemě takovýmto způsobem:[36]

```
[[precondition("x > y")]] int func(int x, int y);
```

Tato funkce by brala parametr typu `Expression`, popisující invariant, který musí platit pro parametry funkce[37]. Protože zná pozici deklarace ve stromě celého programu, je funkce schopná proiterovat celým zbytkem překladové jednotky a najít každý výraz, kde je funkce `func` volaná[38]. Před každý příkaz obsahující takový výraz by vložila nový příkaz ve tvaru `assert(condition);`[39], čímž by se dosáhlo ověřování zadané podmínky při běhu programu v místě volání[40].

Jak bylo uvedeno na začátku kapitoly, metatypy a kód vyhodnocovaný při překladu je nad rámcem práce, vykonané v rámci DP a realizované v překladači, nemá tak smysl

[35] Document Object Model, reprezentace HTML dokumentu jako strom objektů.

[36] V těle funkce je ošetřeno, že kdyby byla umístěna u jiného typu příkazu než je deklarace funcke, napsala by rozumné chybové hlášení a oznámila chybu překladači

[37] Detaily navázání parametrů funkce na idenfikátory ve výrazu ignorujme.

[38] Podobně častá operace by se pochopitelně dala řešit mnohem efektivněji nativně řešeným iterátorem.

[39] Opět, pro zjednodušení výkladu zanedbán krok, kdy se proměnné ve výrazu substituují za skutečné atributy v místě volání.

[40] Pochopitelně, výrazně jednodušší a efektivnější by bylo tento test umístit jen jednou na začátek těla funkce, ale cílem zde bylo demonstrovat expresivní sílu takovéhoto metaprogramovaní.

```
typedef struct pair(T1,T2) {
  T1 first;
  T2 second;
} pair;
...
pair{string,int} p;
```

C+

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
};
...
pair<string,int> p;
```

C++

Obrázek 4.23: Srovnání syntaxe definice obyčejného parametrizovatelného typu v C+ a C++.

zabíhat do větších podrobností. Cílem bylo především naznačit možný systematický přístup k generickému/metaprogramování řešený ve stejném jazyce jako samotný kód, se splněním požadavků kladeným na začátku této části. Náročnost implementace takovéhoto systému se může zdát neakceptovatelná, C+ překladač ale těží z toho, že překládá z nadmnožiny C do C a to pomocí vysokoúrovňových transofrmací, které nevyžadují optimalizace – interní reprezentací programu tak není např. trojadresný kód, ale právě hierarchická stromová struktura výrazů a příkazů. Metatypy by tak nebyly nic jiného, než lehce abstrahované rozhraní na práci s interními daty a datovými typy překladače.

Pro srovnání je třeba zmínit existenci momentálně vyvíjeného programovacího jazyka Jai, který je cílený podobným směrem jako C+ (tedy vysoce výkonný kompilovaný jazyk) a realizoval v překladači zcela plnohodnotný interpret. Přestože myšlenky z této kapitoly byly zformulovány před seznámením se s tímto jazykem, ukázalo se, že řadu věcí řeší podobným způsobem a umožňuje při překladu modifikovat abstraktní syntaktický strom programu, je tedy alespoň zřejmé, že tento přístup je v praxi použitelný. [7]

4.7.4 Parametrizovatelné typy a funkce v C+

Jak bylo řečeno dříve, po funkční stránce jde o zjednodušený systém šablon převzaný z C++. Co se týče syntaxe pro specializaci, jako lepší varianta `List<T>` se jeví `List{T}`. Kromě toho, že netrpí popsanou syntaktickou nejednoznačností, je logickým pokračováním syntaxe pro definici struktury, podobně jak ona definuje konkrétní obsah `struct` tak zde definuje konkrétní případ generického typu. Platí, že pouze vlastní typy (tedy ty definované pomocí `typedef`), mohou být specializované. To odstraňuje konflikt v gramatice, zda je `struct X {...}` definicí struktury nebo už specializací. Kromě toho, pouze vlastní typy mohou mít přetížené operátory a spojené jevy, je zde tak tlak mít všechny „speciální“ datové typy řešené jednotně.

Porovnejme nyní na příkladu syntaxi pro definici jednoduchého datového typu tvořeného ze dvou nespecifikovaných podtypů (4.23). Předně, není nutný prolog ve tvaru `template<...>` deklarující parametry obecné šablony. Protože se typ v C+ dá parametrizovat pouze datovým typem, není tento fakt nutné specifikovat. Seznam použitých typů je v kulatých závorkách za `struct`, resp. za jménem struktury. Bohužel na rozdíl od C++, kde je po definici `struct pair` automaticky definovaný `pair`, v C+ tomu tak kvůli zpětné kompatibilitě s C není. Protože jsou ale specializovatelné pouze vlastní typy, je nutné na definici použít `typedef`. U rekurzivních struktur toto vyžaduje předdeklaraci použité struktury (4.24).

Pro parametrizovatelné funkce je syntaxe lehce odlišná. Stále není nutný prolog, parametrizovatelné funkce lze idenfitikovat použitím typového specifikátoru `auto`. Ten má

```
typedef struct List(T) List;
struct List(T) {
  List{T} *next;
  T data;
};
...
List{int} l;
```

C+

```
template <typename T1, typename T2>
struct List {
  List<T> *next;
  T data;
};
...
List<int> l;
```

C++

Obrázek 4.24: Srovnání syntaxe definice rekurzivního parametrizovatelného typu v C+ a C++.

v C význam implcitního typového specifikátoru pro lokální proměnné ve funkci, z tohoto důvodu je prakticky jen historický artefakt. C++ mu v C++11 přiřadilo význam nový, zpětně nekompatibilní – automatické odvození typu v deklaraci. V C+ se tato funcionalita neplánuje, takže `auto` může být aplikováno i na funkce, kde v C nemá žádný význam.

Jednotlivé parametrizovatelné datové typy *není třeba deklarovat předem*, syntaxe je upravena tak, aby se ve všech výskytech (funkční parametry a typové specializace) dali prefixovat klíčovým slovem `typedef`, což je nutné u prvního výskytu z důvodů syntaktické analýzy. Příklad 4.25 demonstruje rozdíl syntaxe.

Speciální případ je ten, kde je parametrický typ návratovým typem funkce, neboť v tomto kontextu nelze napsat `typedef T`, zejména proto, že v tomto kontextu je výskyt `typedef` validním a vede na konvenční definici datového typu. Proto je tento případ řešen zvláštní syntaxí `auto{T}`, kde se takto „deklaruje“ budoucí parametr a zároveň je považován za návratový typ funkce – příklad je např. `auto{T} Queue.pop(Queue{typedef T} &self) {...}`. Bohužel, v rámci bakalářské práce byl během raného návrhu přetěžovaní učiněn chybný předpoklad, že není smysluplné povolit mezi přetíženími odlišné návratové typy[41] a pro tento předpoklad byla optimalizována interní reprezentace přetížení. Vzhledem k nízké prioritě nebyla tato chyba v návrhu zatím odstraněna, generické funkce vracející parametrický typ jsou tak momentálně limitovány na 1 instanciaci, což však stačí na demonstraci principu.

Generické funkce jsou také implicitně `mangled` z pochopitelných důvodů (viz 5.1).

4.7.5 Problém nárůstu kódu

Využití šablon vede ke generování velkého množství kódu, cílem této části je zmínit možné přístupy, jak tento problém redukovat.

Pokud by naším cílem bylo zredukovat počet instancí na jedinou, jediným řešením by bylo řešit generické funkce na principu rozhraní, podobně jak je chápe jazyk Go – tam je proměnná typu „rozhraní“ ve skutečnosti dvojice (typ, reference na hodnotu). Konkrétní rozhraní I má definovanou množinu metod, co musí splňovat, při známém datovém typu se pak dá dohledat v pomocných strukturách mapování (Typ, Rozhrani) -> Funkce. Co je však zajímavé, je zjištění, že tato idea bude fungovat i bez podpory RTTI[42], stačí místo položky Typ rovnou posílat referenci na tabulku metod pro konkrétní typ. Metodou rozhraní se v tomto kontextu myslí funční ukazatel, kterému se předá reference na hodnotu (efektivně `void *`). Ačkoli je tento přístup funční pro jednoduché příklady, je v praxi příliš

[41] Nepočítaje void a ne-void subvarianty pro stejné argumenty.
[42] Run-Time Type Information – informace o datových typech za běhu programu.

```
auto void filter(typedef T array[],size_t size,bool (*func)(T),T nullval)
{
  for (size_t i~= 0; i~< size; i++) {
    if (!func(array[i]))
      array[i]=nullval;
  }
}

auto void map(typedef T array[],size_t size,T (*func)(T))
{
  for (size_t i~= 0; i~< size; i++) {
    array[i]=func(array[i]);
  }
}
```

C+

```
template <typename T>
void filter(T array[],size_t size,bool (*func)(T),T nullval)
{
  for (size_t i~= 0; i~< size; i++) {
    if (!func(array[i]))
      array[i]=nullval;
  }
}
template <typename T>
void map(T array[],size_t size,T (*func)(T))
{
  for (size_t i~= 0; i~< size; i++) {
    array[i]=func(array[i]);
  }
}
```

C++

Obrázek 4.25: Srovnání syntaxe parametrizovatelné funkce v C+ a C++.

omezený. Bez RTTI nelze z takto zapouzdřené hodnoty nijak vydedukovat původní typ, tedy operace jsou skutečně omezené na ty v rozhraní. Kromě toho, tím, že jsou data předávaná referencí (resp. ukazatelem), mohou být předávána přes zásobník pouze směrem dolů, nelze snadno vracet hodnotu typu rozhraní. Závěrem tedy je, že tento přístup není pro tento jazyk efektivní.

Druhým přístupem je použití void ukazatelů, jak úspěšně demonstruje např. C knihovní funkce `qsort`. Bohužel, `void *` není typově bezpečný a nekonkrétnost odkazovaného datového typu snižuje čitelnost. Pro účely optimalizace nás tak nezajímají přímo funkce využívající `void *`, ale právě takové parametrizované funkce s typem T, kde se se samotným T nikdy nepracuje (ať přímo nebo v podfunkci), neboť pro všechny specializace takové funkce by stačila jediná implementace – příkladem takové funkce by mohl být výpočet délky jednosměrně vázaného seznamu `List{T}`. I když však jde o lákavou myšlenku, patrně neexistuje dostsatečné množství funkcí, aby mělo smysl toto řešit.

Jednoduchým, ale efektivním řešením by bylo přidání klíčového slova `explicit`, které by povolovalo pouze manuální instanciaci parametrizované funkce. Problém implicitní instanciace je, že má tendenci vytvářet nové instance, kdykoli je to možné. Pokud již máme instanci `int func(int)` parametrické funkce `int func(T)`, tak bychom pro argument typu `short`, nebo i `const int` často raději preferovali tuto volbu, při implicitní instanciaci se ale vytvoří instance pro každý případ zvlášť. Explicní kontrola a přehled nad existujícími instanciacemi tak za cenu menší přídavné práce mohou výrazně redukovat nárůst kódu.

Konečně, instanciace jsou definovány lokálně v každé překladové jednotce. Toto je velmi neoptimální, ale jde o typický problém, který se řeší podporou v linkeru, což není dostupné pro překladač překládající do programu v C modul po modulu.

4.7.6 Rezoluce přetížení a typová unifikace

Mechanismus, jakým je modifikována rezoluce přetížení pro implicitní instanciaci parametrizovatelných funkcí, je následující:

1. Pokud je přetížení X pro danou funkci generické a není explicitní, proveď následující:
   (a) Pro každý argument funkce proveď typovou unifikaci mezi reálným typem a odpovídajícím typem v signatuře generické funkce. Výsledkem je množina substitucí parametrů za reálné typy.
   (b) Pokud není množina substitucí úplná či se typy unifikovat nepodařilo, toto přetížení není aplikovatelné.
   (c) Vygeneruj signaturu instanciace pro zadanou množinu substitucí.
   (d) Pokud už existuje přesně stejné přetížení, toto přetížení není nutné.
2. *Běžný kód rezoluce, který se snaží určit jediné nejlepší přetížení. Generické funkce jsou v porovnávání nahrazeny konkrétními substituovanými variantami.*
3. Pokud je výsledná zvolená funkce instanciací, naplánuj instanciaci zvolené specializace. To mimo jiné znamená, že je deklarace této funkce vložena před kód, kde je poprvé využívana – pro další rezoluci přetížení je tedy pak tato specializace chápána jako jakákoli jiná funkce.[43]

[43] Jak bylo řečeno, neuvažuje se SFINAE, selháni instanciace těla funkce je chybou.

Typová unifikace v kontextu C+ je proces, kde porovnáním parametrizovaného typu (`T *`) s konkrétním (`int *`) odpovídáme na otázku, jaký typ je T, resp. zobecněně hledáme pro množinu parametrů konkrétní substituce. Lze ji řešit řadou způsobů, ve funkcionálních jazycích je dost populární algoritmus na Hindley-Milnerově systému (např. v Haskellu), uplatnil se ale třeba i v imperativních jazycích Rust a Swift. Pro potřeby tohoto jazyka však byl využit výrazně omezenější, striktně jednoprůchodový algoritmus.

Algoritmus dostane dva typy (jeden potenciálně generický a druhý reálný) a zpracovává je po jednotlivých úrovních indirekce (od pozice, kde je v deklaraci identifikátor, k typovému základu). Aby byly typy inifikovatelné, jednotlivé indirekce si musí odpovídat typem i kvalifikátory. Při indirekci funkcí se na seznam parametrů funkce unifikace volá rekurzivně. Po zpracování indirekcí nakonec v generickém typu zůstane jen typový základ. Pokud nejde o parametr, musí odpovídat základu druhého typu. Pokud jde o parametr, substitucí je to, co zbylo ve druhém typu; všechny substituce jednoho konkrétního parametru však musí být vždy stejné.

Popsaný algoritmus funguje dobře na případy, kdy se unifikují dva stejné typy, které by měly být striktně stejné, při použití v rezoluci přetížení ale mohou nastávat i situace jako „L-hodnota typu `int`“ vs „T &“, pro tyto účely je tak vhodné odstranit reference a kvalifikátory nejvyšší úrovně indirekce.

V rámci rezoluce přetížení také mohou nastat situace, kdy „generickým typem“ bude třeba `size_t` vůči reálnému typu `int`. Obecně tedy pokud porovnáváme části datatypu, ve kterých se nevyskytuje parametr, unifikace je úspěšná, pokud jsou kompatibilní. Vzhledem k flexibilitě konverzí v C jsou však kompatibilní i typy jako `int` a `char *`, pokud bychom však zkoušeli unifikovat `int` a `T *`, nejsme schopni určit T.

Konečně, nejzávažnější problém je spojen s typovými kvalifikátory. Uvažme, že `char const *` a `char const const *` jsou oba povolené zápisy stejného typu. Pokud porovnáváme `char const *` s `T *`, pak je T jistě `char const`, pokud ale porovnáváme `char const *` s `T const *`, není jasné, zda je T typ `char` či `char const`. Naštěstí, tento problém je řešitelný tím, že při unifikaci nechápeme substituci jako jediný typ, ale jeho nejvyšší kvalifikátory uchováváme jako dvojici nejnižší-nejvyšší možná kombinace. Pokud na konci unifikace máme stále více, než jednu možnost substituce, můžeme si dovolit brát spodní odhad.

Uvažme následující příklad.

'T (T volatile)' vs. 'char const (char const volatile)'

Nejprve unifikujeme parametr funkce, dolní odhad T je `char const`, horní je `char const volatile`. Unifikací s druhým T zjistíme, že je možnost `char const volatile` není možná a odhad zpřesníme. Podobně pro další příklad, kde ale výsledný typ musí být `char const volatile`.

'T (T volatile)' vs. 'char const volatile (char const volatile)'

Poznámka. V překladači byl pro jednoduchost implementován slabší systém, který vždy předpokládá dolní odhad.

4.7.7 Instanciace

Instanciace parametrického typu pouze vytvoří definici s konkrétní specializací, každá další instance se stejnými typy referuje na stejnou definici. Instanciace funkce má oproti C++ unikátní problém v tom, že přetížení operátorů/metod není součástí definice typu, ale může

být dodefinováno později[44]. Proto by dvě totožné instanciace, avšak umístěné v různých místech kódu, mohly vypadat odlišně. Pro jednoznačnost tak není instanciace *definována* před místem prvního použití, ale jen deklarována. Těla všech instanciací jsou pak vygenerována až na konci modulu, kde jsou všechny existující funkce známy. Pro jeden soubor substitucí se tak instanciace provádí nejvýše jednou.

Samotná instanciace vypadá konceptuálně takovým způsobem, že se proiteruje tělo i hlavička parametrizovatelné funkce a každý výskyt parametrizovatelného typu se nahradí substitucí. Potom se udělá (opožděná) sémantická analýza, která např. u funkcí zvolí použitá přetížení.

Sémantická omezení na použití datových typů z této části jsou následující. Parametrické typy bez specializace se nemohou vyskytovat prakticky nikde, vyjímkou je typ složených identifikátorů, lze tak udělat metodu sdílenou mezi instancemi. Specializující typy nesmí být referencemi a musí být reprezentovatelné ve schématu zakódování typů do identifikátoru, zejména toto omezení vyřazuje v současné verzi pole (protože jejich velikost je nevyhodnocovaný výraz).

[44]Teoreticky se tento problém mohl v menší míře projevujevit i v C++ kvůli obecnému přetěžování. Jádrem problému je nejasnost, vůči kterému místu brát soubor přetížení k rezoluci – k definici šablony nebo k místu její instanciace (nejednoznačné).

Kapitola 5

Práce na překladači

V této části jsou stručně nastíněny důležitější práce vykonané na překladači, ne nutně související s hlavními rozšířeními z minulé kapitoly, neboť v celkovém součtu tyto podružné práce zabraly významnou část vývojového času.

5.1 Name mangling

Tímto termínem se označuje proces vytvoření unikátního idektifikátoru pro každé přetížení funkce. Detaily není na tomto místě třeba řešit, důležité vlastnosti formátu jsou však standardizovanost[1] a fakt, že výsledek je stále validní C identifikátor, lze tak takové funkce bez větších komplikací volat a používat stejně jako normální C kód. [19]

Pro zpětnou kompatibilitu s C bylo nutné, aby překladač nedělal mangling zcela automaticky, neboť funkce napsané v C musí být referovány jejich skutečným jménem. V C++ se toto řeší deklarací C funkcí v bloku `extern "C"`. C+ je v návrhu flexibilnější, umožňuje, aby jedna funkce může zároveň mít napsanou implementaci v C a přesto dodávat přetížení v C+. Nejvýše jedno přetížení však může být tzv. *unmangled*, tedy ponechávající si původní jméno.

Překladač může pracovat ve dvou režimech, buď implicitně předpokládá, že jsou všechny funkce přetížitelné, nebo naopak, že je každá funkce `unmangled`. Jednotlivým deklaracím lze explicitně přidat atribut `unmangled` či `mangled`. Mít všechny funkce `mangled` implicitně je výhodnější pro překlad čistého C+ kódu, pokud ale chceme použít C hlavičkový soubor jako `<stdio>`, pak je pro správnou funkčnost nutný druhý režim.

Mangling se vztahuje jen na funkce, netýká se deklarací proměnných a datových typů. Složeného identifikátor tvaru `T.jmeno` je zakódován jako zakódování typu konkatenovaného se jménem. Instanciace parametrizovatelných typů jsou řešeny obdobně jako přetížení, konkatenace jména se seznamem všech zakódovaných specializujících typů.

5.2 Vylepšení vstupů a výstupů

Byly vylepšeny konfigurační možnosti překladače, z příkazové řádky tak lze nově zejména ignorovat jednotlivá varovná hlášení. Kromě toho lze nastavovat formátovací parametry výstupu, jako (měkký) limit znaků na řádek či preferovaný formát umístění levé složené závorky.

[1] Oproti situaci v C++, kde může mít každý překladač formát vlastní, často dochází i k významným změnám mezi verzemi jednoho překladače

Modul realizující výpis byl přepracován pro větší modularitu, kód je více sémanticky orientován a výpis je chápán jako vytváření posloupnosti tokenů oproti filosofii `generování textu`. S tím bylo spojeno zlepšení formátování např. přidáním nadbytečných mezer k významným operátorům jako `=` a odstraněním estetických chyb jako výpis zcela prázdných či nesprávně odsazených řádků, celkový výstup tak (alespoň pro vstup nevycházející z preprocessoru) již vypadá po stránce čitelnosti jako psaný člověkem.

Výstupní modul je natolik abstrahovaný, že by šel potenciálně rozšířit/zaměnit například za kód, který by interní reprezentaci zpracovávaného programu vypsal místo C kódu jako model programu v XML, který by pak šel vhodně vizualizovat v nějakém extérním nástroji.

Velmi významné pro kvalitu překladače je také podávat kvalitní a smysluplná chybová hlášení. Oproti původní verzi je mnoho běžných hlášení informativnějších, obsahují např. jméno funkce, kde nastala chyba.

Bohužel však priotitou bylo v rychlém čase implementovat mnoho nové funkcionality, takže ačkoli je drtivá většina chybových stavů detekována, programátor v jistých případech dostává jen velmi obecný popis chyby. Jedním z důvodů je to, že vnitřní struktury (např. pro příkazy a výrazy) v současnosti neuchovávají jejich pozici ve zdrojovém kódu, mimo fázi syntaktické analýzy tak není způsob, jak sdělit programátorovi, kde nastala chyba.

V budoucnosti by tak bylo vhodné tyto problémy vyřešit systematicky, jak úpravou datových struktur, tak návrhem jednotného formátu hlášení, poskytující všechny relevantní informace. Součástí tohoto by mělo být i vyřešení ztráty čitelnosti způsobené preprocessingem, vhodným kandidátem je preprocesorová direktiva `#line`.

Jedním z nejčastěji se vyskytujících varovných hlášení se týkalo „nebezpečných“ implicitních přetypování, jde o taková přetypování, která v C++ vůbec povolena nejsou, v C ale ano, např. `int` na `char *`. Přetypování mezi různými typy ukazazelů patří do této kategorie, většina těchto hlášení se ale týkala specifického bezpečného případu převodu mezi `void *` a `T *`, jak ho běžně využívá např. funkce `malloc`. Tento případ už v nové verzi za nebezpečný považován není.

5.3 Refaktoring

Pro pokračující vývoj bylo nezbytné vyřešit špatnou udržovatelnost vzniklou duplikací kódu mezi podobnými pravidly při syntaktické analýze.

Prvním takovým případem bylo zpracovávání deklaračních specifikátoru a zpracovávání seznamů specifikátorů/kvalifikátorů. Jde o velmi rozsáhlou část kódu, ale druhý případ je podmnožinou toho prvního, tj. v seznamech specifikátorů/kvalifikátorů se pouze nemohou objevit specifikátory třídy uložení a „funkční specifikátory“, což efektivně znamená jen `inline`. Funkce řešící zpracovávání seznamů specifikátorů/kvalifikátorů byla tedy redukována na volání generičtější varianty s ošetřením nepovolených stavů.

Ještě rozsáhlejší byla série funkcí, analyzující různé typy deklarací (extérní, obvyklá a zapouzdřená v cyklu `for`) a různé typy deklarátorů (obvyklý[2], abstraktní[3] a kombinovaný[4]). Těchto šest funkcí se podařilo sjednotit na dvě zobecněné varianty, nešlo však o triviální úkol, neboť každý z případů se v drobných, ale podstatných detailech liší. Zobecněným variantám tak je dodána informace, jaký konkrétní případ řeší, a v relevantním místě je na tomto základě odchylka ošetřena.

[2] S idenfifikátorem.

[3] Bez identifikátoru, tedy specifikující indirekce datového typu.

[4] Používaný u parametrů funkcí; může, ale nemusí mít identifikátor.

```
//Nejlogickejsi format, pole1 je slozeno ze 3 prvku,
// kazdy je rekurzivne inicializovan ve slozenych zavorkach
int pole1[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
//Tzv. flat inicializace bez zavorek
int pole2[3][3] = {1,2,3,4,5,6,7,8,9};
//Nejednotne uzavorkovani je take povoleno
int pole3[3][3] = {1,2,3,{4,5,6},7,8,9};
```

Obrázek 5.1: Ukázka tří stejně inicializovaných polí s jinám závorkováním initializer listu.

Menšího úspěchu bylo dosaženo v ostranění rendundance u sémantické analýzy, konkrétně kód řešící rezoluci výsledného typu podvýrazu, s vyhledáváním přetížených operátorů. Podařilo se sice sloučit případy patřící do stejné podtřídy (jako všechny relační operátory), obecně ale bylo mezi jedlitlivými případy příliš rozdílů na významnější zkrácení.

5.4 Odladění úniků paměti

Při čištění kódu bylo zjištěno, že části paměti využité při běhu překladače nebyly na konci správně uvolňovány, což je u programování v C reálnou hrozbou. Pomocí nástroje `Valgrind` ale byly všechny chyby postupně opraveny a překladač tak i při překladu rozsáhlých programů nevykazoval žádné ztráty.

5.5 Podpora `__func__`

Speciální identifikátor `__func__` v C99 obsahuje textový řetězec s názvem funkce, praktický pro logovací funkce. Oproti podobným věcem jako `__LINE__` a `__FILE__` ale není definovaný jako makro (a tedy řešený preprocesorem), ale chová se jako (potenciální) skrytá lokální proměnná každé funkce. Podporovat ho bylo nutné mj. proto, že je někdy využíván např. pro implementaci makra `assert`, bez jeho podpory by tak nefungovaly ani standardní knihovny, i kdyby jej programátor sám nevyužíval.

Samotná podpora je řešena jako speciální případ funkce vyhledávající význam identifikátoru – pokud není identifikátor definovaný v současné funkci, je proveden explicitní test na `__func__`, než se přejde na vyšší úroveň.

5.6 Sémantika initializer listu

Konstrukce „initializer list“, primárně využívaná pro inicializátory složených typů a polí, nebyla v předchozí verzi sémanticky testovaná na validitu. Důvodem je vysoká laxnost, kterou zde C povoluje, jak ukazuje 5.1, a s ní spojená složitost vyhodnocování.

Základní idea je, že pokud je očekáván člen inicializující strukturu či pole a uvedená hodnota (nebo initializer list) není daného typu, tak se předpokládá, že je inicializován první prvek struktury či pole. Tento mechanismus je ještě zkomplikován *designátory*, kterými se dá vybrat, který prvek je právě instanciovaný a libovolně tak v hierarchii přesouvat „kurzor“ na aktivní prvek. Tento přístup spoléhá, že neexistuje nejednoznačnost mezi inicializací složeného typu a jeho prvku, jinak se začíná chovat pro programátora nepřehledným

způsobem. Navíc, jak bylo zmíněno dříve, v dlouhodobém plánu jazyka jsou uvažovány přetížitelné inicializátory pro netriviální typy, nekorektní uzávorkování by zde bylo neakceptovatelné.

Z uvedených důvodů je C+ v sémantice initializer listu restriktivnější než C a umožňuje pouze první variantu v příkladu 5.1, tedy plné závorkování.

5.7 Ostatní změny

Do procesu transformace C+ na C bylo třeba zakomponovat všechny zmíněné kroky, jako např. přidání volání destruktorů. Z důvodů efektivnosti bylo vhodné, aby se všechny tyto operace provedly v rámci jednoho průchodu programem, to však vyžadovalo prověření, že se jednotlivá rozšíření navzájem neovlivní neočekávaným způsobem. Naštěstí ale mimo menší problémy jedinou výraznou komplikaci způsobily reference.

Jak bylo zmíněno, reference jsou principiálně implementovány tak, že typ `T &` resp. `T &&` je konvertován na `T * const`, resp. `T * const restrict`. Všechny případy užití reference `x` jsou pak zaměněny za `*x`. V praxi se nejpřív proiterují výrazy a pro každou proměnnou typu reference se proměnná dereferencuje. Teprve potom dochází ke změně datových typů. Jak vidíme, problémem tu je skrytý předpoklad, že se na jeden úsek kódu nepustí konverze vícekrát, zejména pokud je proměnná dosud v nekonzistentním stavu (již je dereferencovaná, ale stále s typem reference). Eventuálně se ukázalo, že v jistých situacích tento předpoklad neplatí, resp. je jednodušší konverzi výrazů vyjímečně zopakovat, než ukládat plošně informace, které výrazy už byly či nebyly zpracovány. To však vedlo k tomu, že reference byly dereferencované opakovaně, neboť je poměrně obtížné tento fakt detekovat. Nakonec bylo zvoleno neelegantní, ale pragmatické řešení, kdy byl čistě interně definován nový operátor „dereference reference“, který se po všech stránkách chová stejně jako obyčejná dereference, jen jeho přítomnost znemožnuje opakovanou dereferenci referencí.

Implementace parametrických funkcí také vyžadovala řadu změn. V rámci zpracování deklarace se postupně generuje seznam generických parametrů, který je potřeba předávat do všech podfunkcí, které jsou schopny přijímat identifikátor a potřebují vyhledávat jeho význam.

Z tohoto důvodu nejsou v parametrických funkcích zcela podporovány implicitní hodnoty parametrů funkce – pro tento úzký případ použití by se musela kromě funkcí zpracovávající deklarace stejným způsobem rozšířit i množina funkcí zpracovávající výrazy, to vše jen, aby se potenciální výskyt generického parametru T ve výrazu správně pochopil.

Kapitola 6

Závěr

Cíle, stanovené v zadání diplomové práce, byly úspěšně splněny.

Do detailů byly prozkoumány a porovnány možnosti populárních, ale i řady méně známých programovacích jazyků. Pro vyvíjený jazyk C+ byla navržena a realizována celá řada nových rozšíření, čerpajících z dosažených poznatků. Co však je důležitejší, kromě dílčích vylepšení bylo v jejich rámci vytvořeno několik originálních přístupů k daným problémům, například v sémantice přesouvání, práci s metodami či systematickému přístupu jazyka ke generickému programování.

Překladač jazyka byl významným způsobem rozšířen a je mnohem více prakticky použitelný, podařilo se totiž dosáhnout kompletní automatizace procesu překladu až ke spoustitelným souborům – oproti předchozí verzi, která pouze umožňovala převádět zdrojové soubory z C+ do C, tak jde o velký pokrok. Kromě toho je teď překladač stabilnější a konfigurovatelnější než byl dříve.

Jazyk si uchovává své cílení na specifickou oblast použití, tedy zkomfortnění práce na úrovni nízkoúrovňového jazyka a zvýšené jeho čitelnosti, rozhodně ne však za cenu kompromisu na výkonnosti. Tímto konkrétním směrem není zaměřeno příliš mnoho jazyků, ty existující však mají tendenci přicházet s experimentální syntaxí a nutit prográmátory učit se novým konceptům. Proto vůči nim jazyk, který je striktní (a minimalistickou) nadmnožinou C, nabízí rozumnou alternativu. Díky nové funkcionalitě, kterou byl posílen, by mohl být relevatní volbou pro mnoho úkolů, kde je použití „většího“ (jako C++) nebo „pomalejšího“ (jako většina iterpretovaných vysokoúrovňových jazyků) jazyka neadekvátní.

Není lehké v této chvíli odhadnout budoucí vývoj projektu. Průzkum spektra programovacích jazyků používaných v současnosti jasně ukazuje, že populární programovací jazyky mají velmi vysokou hybnost a i takový nový jazyk, který je po každé stránce lepší, než jeho předchůdce, nedokáže nahradit zavedený standard jakým je třeba C/C++, což ilustruje např. (relativní) neúspěch jazyků D a Rust. V dnešní době je také konkurence ozvlášť vysoká, protože díky platformě LLVM lze pro zcela nový jazyk relativně snadno napsat překladač, který produkuje dostatečně rychlé a slušně optimalizované programy.

Přes zmíněná fakta je však tato práce jednoznačně smysluplná; celkovým účelem projektu nebylo vytvořit jazyk, který by se snažil vytlačit pozici C nebo Pythonu, ale který poskytuje zajímavou alternativu ve své specifické doméně. I kdyby však zůstal ryze akademickým dílem, tak poznatky, praktiky, a zkušenosti, které se zde objevily, jsou právě takovým zdrojem informací, díky kterým se ony zavedené jazyky pomalu inovují a posouvají tím celé odvětví programovacích jazyků kupředu.

Literatura

[1] Attributes in Clang. [online], [viděno 4. ledna 2017] Dostupné z: clang.llvm.org/docs/AttributeReference.html.

[2] C# language reference. [online], Poslední modifikace: 20. července 2015 [viděno 4. ledna 2017] Dostupné z: msdn.microsoft.com/en-us/library/618ayhy6.aspx.

[3] C99 Rationale. [online], Poslední modifikace: duben 2003 [viděno 14. března 2017] Dostupné z: www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf.

[4] GCC 4.9.4 compiler manual. [online], [viděno 20. prosince 2016] Dostupné z: gcc.gnu.org/onlinedocs/gcc-4.9.4/gcc/.

[5] Go language reference. [online], Poslední modifikace: 31. května 2016 [viděno 4. ledna 2017] Dostupné z: golang.org/ref/spec.

[6] Haskell 2010 Language Report. [online], [viděno 16. května 2017] Dostupné z: www.haskell.org/definition/haskell2010.pdf.

[7] Jai Primer (popis charakteristik vyvíjeného jazyka Jai). [online], Poslední modifikace: 22. dubna 2017 [viděno 15. května 2017] Dostupné z: github.com/BSVino/JaiPrimer.

[8] Java language reference. [online], Poslední modifikace: 13. prosince 2015 [viděno 5. ledna 2017] Dostupné z: docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[9] Microsoft-specific modifiers (MSVC dokumentace, Visual Studio 2015). [online], [viděno 5. ledna 2017] Dostupné z: msdn.microsoft.com/en-us//library/6bh0054z.aspx.

[10] Programming Languages — C++ Extensions for Concepts. [online], Poslední modifikace: 9. února 2015 [viděno 15. května 2017] Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf.

[11] Python 3 language reference. [online], Poslední modifikace: 3. ledna 2017 [viděno 5. ledna 2017] Dostupné z: docs.python.org/3/reference/index.html.

[12] Rust language reference. [online], [viděno 4. ledna 2017] Dostupné z: doc.rust-lang.org/reference.html.

[13] Standard C11 (committee draft). [online], Poslední modifikace: 12. června 2011 [viděno 5. ledna 2017] Dostupné z: www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[14] Standard C++11 (working draft). [online], Poslední modifikace: 28. února 2011 [viděno 5. ledna 2017] Dostupné z: www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[15] Standard C++14 (working draft). [online], Poslední modifikace: 19. listopadu 2014 [viděno 15. května 2017] Dostupné z: www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf.

[16] Standard C++17 (working draft). [online], Poslední modifikace: 21. března 2017 [viděno 15. května 2017] Dostupné z: www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

[17] Standard jazyka C99 – ISO International Standard ISO/IEC 9899:TC3:2011. [online], [viděno 21. listopadu 2014] Dostupné z: www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[18] Dijkstra, E.: Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, ročník 11, č. 3, Březen 1968: s. 147–148, ISSN 0001-0782.

[19] Fog, A.: Calling conventions for different C++ compilers and operating systems. [online], Poslední modifikace: 1. května 2017 [viděno 15. května 2017] Dostupné z: www.agner.org/optimize/calling_conventions.pdf.

[20] Garcia, R.; Järvi, J.; Lumsdaine, A.; aj.: A Comparative Study of Language Support for Generic Programming. [online], [viděno 11. března 2017] Dostupné z: www.osl.iu.edu/publications/prints/2003/comparing_generic_programming03.pdf.

[21] Meduna, A.: *Elements of Compiler Design*. New York: Taylor & Francis, 2008, iSBN 1-4200-6323-5.

[22] Opatřil, P.: Rozšíření jazyka C a jeho překladač. 2015.

[23] Sebesta, R. W.: *Concepts of programming languages*. London: Addison-Wesley, 10 vydání, 2012, iSBN 0-13-139531-9.

[24] Stroustrup, B.: *Standard C++*. Chichester: J. Wiley, 2003, iSBN 0-470-84674-7.